UNIVERSIDAD CARLOS III DE MADRID

Escuela Politécnica Superior

Ingeniería Informática

Proyecto Fin de Carrera

Jaminion: **Plataforma de Dominion**

Autor: D. Sergio Durán Vegas

Tutor: Prof. Álvaro Torralba Arias de Reyna

Julio, 2013

Contenido

Índice de ilustraciones

Índice de tablas

Introducción

El Dominion es un juego de mesa de estilo Europeo creado por Donald X. Vaccarino, este fue lanzado en 2008 y ha sido uno de los títulos más importantes del sector de finales de la pasada década.

Los juegos Europeos a diferencia de los americanos se basan más en una organización estratégica y económica que en el azar o la competitividad, además la interactividad entre jugadores pasa ser indirecta y normalmente se reduce el número de conflictos así como se mantiene a todos los jugadores hasta el final de la partida. Las reglas son más simples y las partidas no suelen alargarse más de una o dos horas. Que las reglas sean más simples hace posible que se pueda diseñar tanto un simulador como una IA totalmente fidedignas al juego real.

En los últimos 10 años, la inmersión de la población en internet es cada vez más acusada y a través de más medios, como teléfonos, ordenadores y tablets. Esto ha facilitado la conversión de los clásicos juegos de mesa a formatos digitales mucho más manejables, accesibles y que facilitan al usuario la posibilidad jugar con otras personas sin tener la necesidad de encontrarse en el mismo espacio físico. Esta situación ha sido aprovechada por desarrolladores para lanzar sus propios productos a usuarios a través de portales web o plataformas de aplicaciones como Android Market.

Además, los investigadores han encontrado un criadero de plataformas para poder desarrollar teorías y realizar estudios relacionados con la IA. Muchas veces los productos software que se lanzan al público en general no admiten o no están adaptados para ser modificados o para poder ser usados en el campo de la investigación. Por ello no es raro encontrar proyectos que consisten en adaptar los productos actuales para ser usados en investigación o para admitir modificaciones profundas de funcionamiento por usuarios avanzados.

Volviendo a nuestro juego en particular, Dominion es considerado el primer juego de estrategia en el que el jugador construye su mazo de forma planificada o estratégica a lo largo de la partida. Aun así y conservando la mecánica original, existen nuevas expansiones del juego que lo dotan de nuevas funcionalidades, creando diferentes variantes del mismo juego haciéndolo a su vez más complejo e interesante. En el juego básico, un grupo de entre 2 o 4 jugadores que cuentan con unas cartas iniciales predefinidas deberán construir su mazo, diseñando una estrategia de compras y acciones para que al finalizar la partida tengan el mayor número de puntos de victoria posibles en el mazo.

Nuestra idea principal fue realizar un módulo nuevo sobre un simulador/plataforma existente, como ORTS[1], de Dominion u otros juegos de cartas que pudiera utilizar inteligencias artificiales deductivas que fueran evolucionando a lo largo del juego; de manera que su capacidad de raciocinio no se limitara a una inteligencia reactiva implementada con unos simples if else anidados, si no que pudieran desarrollar estrategias similares a las que idearía una mente humana.

1 ORTS ”Open Real-Time Strategy” es un entorno de programación para la investigación de problemas en tiempo real

Sin embargo una vez exploradas las diferentes alternativas ya existentes en el mercado no encontramos nada que satisficiera nuestros requerimientos o al menos de una manera aceptable, sin tener que dedicar un tiempo bastante largo en adaptar la plataforma a nuestros intereses.

Dada esta situación optamos por crear una plataforma totalmente nueva que cubriera está vacío. Con la intención en este proyecto de crear un juego lo más flexible posible, que no solo diera un gran abanico de posibilidades a los usuarios, sino también a investigadores, dotándolo de una interfaz de muy fácil uso adaptable a IAs.

Con esta plataforma abrimos un importante camino para Dominion tanto en el campo de la investigación como de la experiencia de usuario. Quedarán muchas líneas de mejora abiertas así como varias posibilidades de personalización.

1 Estado del arte

Este capítulo presenta el Dominion como un juego de mesa moderno de estilo Europeo, además de explicar de manera sucinta cómo funcionan estos juegos y en concreto Dominion, intentaremos analizar la relación en general de este tipo de juegos con la informática.

Además, para profundizar en el estado del arte estudiaremos qué productos o proyectos existen en la actualidad relacionados con Dominion tanto a nivel usuario como para desarrolladores o investigadores.

1.1 Introducción a los juegos de mesa europeos

También conocidos como de estilo alemán, estos juegos representan un cambio con respecto a la filosofía americana basada en el azar, en la confrontación directa y en juegos complejos o de larga duración.

- El sistema o la mecánica de juego es más importante que la propia temática del juego.
- Fomento de la participación y cooperación entre jugadores. Se promueve la no-eliminación o la clara diferenciación entre ellos a lo largo de la partida.
- Partidas pensadas para ser jugadas de una sola vez, duración de un máximo de 3 horas.
- Se minimiza el uso de texto escrito (excepto en las reglas), sustituyéndolo en lo posible por símbolos y dibujos, y no se suele requerir el uso de la palabra hablada. Esto facilita la internacionalización del mismo.
- En términos generales, sencillos de jugar o modelar, que no por ello dominar a la hora de jugar.

1.1.1 Historia

Podemos identificar el nacimiento del estilo en los años 60 aunque no fue hasta finales de los 70 y principios de los 80 cuando empezó realmente este movimiento de diseño de juegos gracias a que diferentes grandes compañías del sector invirtieron en el modelo. Sin embargo, cuando realmente se produjo un cambio en el sistema y estos juegos empezaron a ocupar la posición que ahora ostentan fue a mediados de los 90 con el nacimiento de "Colonos de Catán"[2]. Fue lanzado en 1995 y vendió las primeras 5.000 copias tan rápido que el autor no tenía en mente ni lanzar la primera edición oficial (1). Ese año ganó el *Spiel des Jahres* [3] y todos los demás grandes premios del país. Los expertos lo consideraron una obra maestra y tal ha sido que desde su lanzamiento ha vendido 15 millones de copias y abrió el camino que posteriormente seguirían juegos como Puerto Rico, *"Ciudadelas"* o el mismo *Dominion*.

[2] (*Die Siedler von Catan en alemán*) es un juego de mesa multijugador inventado por Klaus Teuber.
[3] (*Game of the year en inglés*) es un premio para los juegos de mesa que premia la calidad en el diseño de juegos.

1.1.2 Actualidad

Hoy en día y tras casi 20 años de expansión, los juegos de estilo europeo han colonizado el mundo, poco a poco el modelo americano, más visual y complejo ha ido perdiendo terreno frente a la sencillez y a la vez dinamismo del modelo Europeo. El avance de juegos como Catán o Puerto Rico no solo se ha expandido por toda Europa si no que ha colonizado América e incluso Asia.

Podríamos decir que este avance es debido a que el modelo de jugador ha cambiado mientras que antes era solo una parte de la sociedad la que se dedicaba a los juegos de mesa u ordenador, ahora se intenta incluir a toda la familia en la cultura de los juegos. Véase el fenómeno Wii, este movimiento lúdico-social ha impulsado a los grandes distribuidores a invertir en este "nuevo" modelo, con fantásticos resultados. Hay que añadir que además de que actualmente se busque un modelo en el que participe toda la familia, hay que constatar que este movimiento también se beneficia de no ser un estilo alternativo, de la típica fantasía medieval o del futuro, o al menos lo suficiente para que sea mal visto por la masa social, con lo que hoy en día ha entrado dentro del *"mainstream"* social en lo que refiere a juegos de mesa.

Estos juegos basados en reglas más sencillas de modelar matemáticamente, han impulsado los estudios de IA basados en este tipo de juegos de mesa, no es difícil encontrar proyectos o estudios universitarios relacionados con Catán (2), Puerto Rico[4], Carcassone[5].

1.2 Dominion

Este es el juego de mesa en el que hemos basado el proyecto, como dijimos en la introducción pertenece al modelo de juego de mesa europeo o alemán, mucho más estratégico que los americanos. Su mecánica es algo diferente con respecto a otros juegos del modelo alemán, ya que guarda similitudes con el juego *Magic: The Gathering,* que es un juego de cartas coleccionables en el cual las cartas definen el juego a través de los efectos que realizan el mismo al ser jugadas. Aun así ambos son diferentes. Dominion no es un juego de cartas coleccionables, en él cada jugador va creando su propio mazo de cartas según avanza la partida. El objetivo final es tener el máximo número de puntos que se obtienen al poseer unas cartas determinadas. Dichas cartas son las conocidas como cartas de territorio que podremos ir comprando durante el desarrollo de la partida. Dado que estas no tienen ninguna utilidad durante el juego salvo esa, el objetivo relativo del jugador irá cambiando a lo largo de la partida. Si bien el objetivo inicial será construir un mazo que le provea de los recursos suficientes, más adelante y una vez obtenidos dichos recursos el objetivo primordial será ir adquiriendo cartas de terreno que le den la victoria al finalizar la partida, dejando en un segundo plano pero sin olvidar, el seguir aumentando los recursos propios del jugador adquiriendo cartas de acción o de tesoro.

[4] http://www.tropiceuro.com/puerto-rico-evolver/
[5] http://jcloisterzone.com/en/download/

1.2.1 Reglas de juego

Dominion es un juego de una duración media de 30-45 minutos durante el cual un grupo de 2 a 4 jugadores competirán por conseguir el mazo con más puntos de victoria. El mazo se crea comprando cartas gracias a los tesoros que jugamos o a través de cartas de acción que nos permitan obtener cartas.

1.2.1.1 Cartas

El elemento más importante de este juego y que realmente lo hace diferente de otros son sus cartas. El juego está totalmente definido por las cartas que contiene y los efectos que ejecuta cada carta en el juego. La carta está definida por nombre, coste y tipo. Hay tres tipos de cartas que dependiendo de cuál sea la carta se jugará en un momento u otro del turno. A la vez que otras contarán de una manera u otra al final de la partida, el tipo de la carta podremos encontrarlo en la parte frontal inferior de la misma.

- Victoria: Son las cartas que al tenerlas en el mazo nos otorgarán puntos de victoria al finalizar la partida. Pueden suponer un problema ya que no tienen efecto durante el juego, se compran siempre a partir de la mitad de la partida.

- Acción: Este tipo de cartas se juegan al principio del turno, durante la fase de acción. Cada una tiene una serie de efectos asignados que varían según la carta. Se las diferencia en dos tipos de manera "extra-oficial". El tipo *"Village"* que al jugarla le otorga al jugador más acciones que jugar aparte de otros efectos y el tipo Terminal que son efectos concretos que no aportan más acciones.
    - Ataque: Tipo especial de carta de acción que aplica unos efectos ofensivos en los jugadores enemigos de la partida.
    - Reacción: Es un tipo especial de carta de acción que se puede jugar para bloquear un ataque de otro jugador. El momento de jugarlo es cuando el jugador ejecuta la carta de ataque.

- Tesoro: Este tipo de carta es la moneda del juego y podremos adquirir más durante el mismo comprándolos, se juegan en la fase de compra y podremos jugar tantos como queramos antes de entrar en la fase de compra.

Las cartas serán accesibles en el juego a través de los suministros, dichos suministros poseen un número limitado de cartas de un mismo modelo. Por ejemplo, habrá un suministro para la carta de *Cobre*, otro para la de *Ducado*, etc…

1.2.1.2 Partida

Antes de iniciar la partida, los jugadores escogerán 10 cartas diferentes de reino de entre todas las posibles. En nuestra plataforma dicha elección de cartas se realiza mediante una plantilla de juego seleccionable antes de cada partida y que definirá que cartas de cada tipo son las que serán accesibles durante el juego y pasarán a ser suministros junto con las de tesoro y victorias; estas cartas podrán adquirirse durante el desarrollo de la partida. Además al iniciar la partida cada jugador recibirá 10 cartas, de las cuales serán 7 de tesoro (Cobre) y 3 de victoria o terreno (Finca). Una vez obtenidas y barajadas el jugador robará las 5 cartas con las que empezará a jugar en su turno.

En cada turno de un jugador tendremos 3 fases diferenciadas, la fase de Acción, la de Compra y por último la de Mantenimiento:

- Acción: Durante esta fase solo se pueden jugar cartas de acción, sean de ataque o normales. Por defecto un jugador dispone de una acción por turno.
- Compra o Tesoro: En esta fase jugaremos los tesoros que queramos de nuestra mano y podremos adquirir alguna carta disponible de las pilas de reino. Por defecto un jugador solo tiene una compra por turno.
- Mantenimiento: En esta fase el jugador descarta las cartas jugadas y las de su mano y roba otras 5 para su próximo turno.

La partida finaliza cuando se han consumido en su totalidad al menos 3 suministros, o se haya acabado el suministro de cartas de Provincia. Una vez finalizada la partida se recuentan los puntos de victoria que posee cada jugador y en caso de empate a puntos ganará el que menos turnos haya jugado.

1.2.2 Modos de juego

Dominion es un juego de mesa y por lo tanto se juega mayoritariamente en persona. Es un juego rápido y todos los jugadores se sientan alrededor de una mesa o similar donde poder jugar las cartas. Es por ello que cuando nos referimos a modo de juego únicamente haremos mención a cómo puede variar el desarrollo de la partida dependiendo de la elección de unas cartas de reino u otras.

En relación con el número de jugadores irá el número de cartas disponible por cada suministro. Es por ello que el desarrollo del juego cambia con respecto al número de jugadores ya que, en una partida de 4 jugadores el promedio de cartas por jugador baja bastante, con lo que la partida será más rápida y habría que estar más atento a las cartas disponibles en los suministros.

Aun así, lo que realmente condicionará el juego será las cartas de reino que se elijan para la partida. Es un juego muy propicio a hacer combinaciones entre cartas, no solo entre las que robas y puedes jugar, si no en las que puedes obtener de tu propio mazo gracias a un efecto de una carta jugada. Por lo tanto el modo de juego estará muy condicionado no solo por las cartas que se puedan adquirir, sino además por las combinaciones que se puedan hacer la jugarlas, tanto las que juegue el propio jugador, como las que puedan jugar los adversarios.

1.2.3 Expansiones

Hemos querido hacer mención especial a este punto porque estará estrechamente relacionado con uno de los objetivos de nuestro proyecto. La reutilización o visto de otra manera, la facilidad de expansión del proyecto para poder abarcar las ampliaciones del juego que sean lanzadas con posterioridad.

Las expansiones de Dominion no solo incluyen más cartas reino, si no efectos nuevos, que normalmente están relacionados entre ellos. Podemos decir que dichas expansiones son expansiones temáticas, que pueden incluir una gama de efectos nuevos que se suman a los que vienen en el juego original ya sean de acción, tesoro o el momento en el que toman efecto una vez han sido jugados.

En definitiva, una expansión significa una variación con respecto al juego original, que normalmente está relacionada con una temática concreta (“Alquimia”,”Cornucopia”,”Intriga”).

1.3 Juegos de mesa digitales

Se considera un juego de mesa digital a todo aquel juego de toda la vida que haya sido adaptado a las nuevas tecnologías. El primero a señalar es por supuesto el Ajedrez, considerado el juego de mesa más importante de la historia; tanto que es el deporte nacional de algunos países. Otros juegos clásicos también han sido digitalizados y estudiados como el Backgammon, Reversi, Conecta cuatro o Dominó. Estos juegos clásicos han sido ampliamente estudiados así como comercializados para el consumidor además de haber sido usados en investigaciones de IA. En general, todos los juegos de mesa acaban siendo digitalizados ya que en el mundo en el que vivimos hacer lo contrario tendría poco sentido, no solo comercial sino también cultural. El mercado y la sociedad demandan cada vez más horas de entretenimiento delante de sus dispositivos multimedia y digitalizar un juego, ya sea clásico o moderno no supone mucho tiempo si lo comparas con el beneficio económico posible.

Ahora sí, analizándolo desde la perspectiva de un investigador, estos juegos clásicos son sencillos de modelar matemáticamente a la hora de hacer un estudio ya que son deterministas y la información de la partida está bastante acotada. Esto difiere bastante de Dominion, ya que la información a la que tendrán que tener acceso los desarrolladores es bastante más compleja que si estuviéramos hablando de alguno de estos juegos clásicos como podemos ver en el estudio (3).

En este apartado haremos un breve estudio sobre la IA relacionada con este tipo de juegos además de repasar las aplicaciones más famosas basadas en este tipo de juegos, sean para usuarios, simuladores para investigación o ambos al mismo tiempo.

1.3.1 Inteligencia Artificial

Uno de los cambios más importantes cuando se produjo la evolución del formato físico al digital fue la introducción de inteligencias artificiales que hicieran posible que una persona probara sus habilidades directamente contra la máquina.

Más allá de convertirse en un simple entretenimiento para el consumidor, estos juegos han ido evolucionando también de cara al investigador, convirtiéndose muchas veces en simuladores para probar Inteligencias Artificiales. A la hora de hablar de una IA que tome parte en un juego, es necesario diferenciar los modelos con los que se trabaja actualmente, analizando sus pros y sus contras, tanto de cara al usuario como al investigador.

En el universo de las inteligencias artificiales podemos distinguir 2 tipos: uno es la inteligencia artificial convencional basada en métodos simbólico-deductivos, basados en el comportamiento humano ante diferentes problemas. El 2º tipo es la inteligencia artificial, llamada inteligencia computacional o inductiva. Es de creación más reciente e introduce un concepto diferente al de la filosofía clásica de la heurística y los sistemas formales. De entre todos los modelos que permiten plasmar una inteligencia artificial, los siguientes son quizás lo más relevantes o más recurridos en el campo del diseño de IAs.

- **Redes Neuronales:** Se componen de un conjunto de unidades llamadas neuronas las cuales reaccionan ante unos impulsos (entradas numéricas) y que da una salida numérica también a partir de 3 funciones diferentes (propagación, activación y transferencia). Estas redes tienen un proceso de entrenamiento por el cual aprenden a clasificar información de la cual se sabe la respuesta, premiando las respuestas acertadas y castigando las erróneas. Tras este proceso de entrenamiento se consigue que estas redes consigan clasificar cualquier hecho representado numéricamente. Incluso con estas redes se pueden crear bots más "humanos" que los propios humanos. (4)
- **Máquina de vectores soporte:** Este sistema estrechamente relacionado con las redes de neuronas es básicamente un clasificador lineal, que a través de un hiperplano separa 2 regiones de puntos dependiendo de un atributo o variable predictora; siendo aquellos puntos más cercanos al hiperplano lo que conocemos como vector soporte. Es un sistema con ciertas ventajas con respecto a las redes neuronales tradicionales, la principal es su facilidad para entrenar el modelo. Cada vez se aboga más por el uso de este modelo como solución a las IAs de los videojuegos en detrimento de las básicas redes de neuronas (5).
- **Sistemas difusos:** Su estructura está constituida por tres bloques principales: el de transformación de los valores numéricos en valores de lógica difusa; el motor de inferencia que emplea las reglas; y el bloque de conversión de los valores de la lógica difusa en valores numéricos. Es un sistema que se ha usado normalmente para sistemas electrónicos pero se pueden encontrar usos de todo tipo como en diagnósticos médicos (6) o incluso en recursos humanos (7) pero además ha sido implementado en videojuegos de manera exitosa como en "Civilization", "Unreal", y "Los Sims".
- **Árboles de búsqueda:** Es un árbol de decisión cuyos nodos representan diferentes estados de la partida, partiendo desde el punto inicial, teniendo tantos hijos o conexiones como posibles variantes existan al realizar un movimiento/acción. Este tipo de árboles suele ir ligado al algoritmo MinMax, que es un algoritmo recursivo que escoge tú mejor opción contando que el adversario escogerá la opción que menos te beneficie, se usó como base para la inteligencia de la famosa máquina de ajedrez de IBM Deep Blue (8).
- **Monte Carlo Tree Search:** Es un método de búsqueda de decisiones óptimas en un dominio concreto. El método consiste en obtener resultados haciendo búsquedas aleatorias y creando un árbol de búsqueda de acuerdo con los resultados obtenidos. Este método ya ha tenido un profundo impacto en el diseño de AIs relacionadas con modelos que pueden ser representados como árboles de decisiones secuenciales. Este modelo se ha probado en diferentes juegos, tanto clásicos como adaptaciones de juegos de mesa modernos, con resultados bastante satisfactorios. Comparado con el clásico modelo de aproximaciones heurísticas, por ejemplo en "Colonos de Catan" está probado que algoritmos basados en MonteCarlo son más potentes que los tradicionales modelos heurísticos o basados en *scripting* (2) .

1.3.2 Aplicaciones populares

De todos los grandes juegos clásicos o modernos existen infinidad de adaptaciones o conversiones, enfocadas a un público u otro. En este estudio diferenciaremos dependiendo del público al que se enfoque dichas plataformas, haciendo un análisis del estado del arte apto para que el lector pueda contrastar debidamente las diferentes opciones que existen.

En principio las primeras adaptaciones no solo se transmitían en formato físico sino que además debían ser previamente instaladas o almacenadas en el disco duro local para poder ser ejecutadas. Este formato copó durante más de 2 décadas todo el panorama de los videojuegos hasta la llegada de internet y sobre todo de la banda ancha que, aumentando la capacidad de descarga y reduciendo la latencia, hizo posible los clientes web ligeros de almacenamiento temporal.

1.3.2.1 Usuarios

Aquí haremos un análisis de las plataformas, enfocadas al usuario más populares que existen. En este análisis haremos bastante hincapié en sus puntos fuertes de cara al usuario así como sus carencias más significativas, con el objetivo de contrastar la calidad de nuestro proyecto.

1.3.2.1.1 Clientes ordenador

Fueron los primeros en aparecer en el mundo digital. En los 90 tuvieron bastante auge los juegos basados en el Ajedrez, los cuales incorporaban inteligencias artificiales basadas en fuerza bruta que resultaban de bastante utilidad para el público general. Algunos expertos afirman que estas plataformas han entrado en desuso, esto no es del todo cierto, ya que dicha afirmación está basada en un análisis que solo toma en consideración ordenadores comunes. Dicho descenso en el uso en plataformas clásicas es consecuencia directa del traslado del sector al mundo de los móviles o tablets que cada día disponen de mayor potencia tanto gráfica como de procesamiento, haciendo posibles atractivos juegos con inteligencias artificiales competitivas. Por lo que podemos afirmar que los clientes *"offline"* siguen siendo uno de los productos importantes del mercado

- Ajedrez: Piedra angular de los juegos de mesa digitales.
    - ChessMaster: Su última versión fue lanzada hace exactamente 6 años, lo que nos puede dar una idea del incipiente abandono que sufren las plataformas clásicas o al menos el software residente a favor de los alojados en la nube. Este juego además de haber sido testeado por grandes jugadores siempre tuvo una gran acogida por el usuario medio. Con una interfaz muy agradable así como un modo de enseñanza o aprendizaje para jugadores de diferentes niveles, se ganó durante años al gran público. Además, desde hace bastante tiempo encontró la manera de atraer al usuario medio-avanzado mediante un sistema de diseño de IA bastante sencillo que daba la posibilidad de aplicar personalidades, así como cambiar el peso relativo [6]de las piezas. Como conclusión decir que si bien no llega ser tan inteligente como otras opciones de la categoría, si es el más amigable de cara al usuario medio

[6] Valor de la pieza en un algoritmo matemático

además de que hay que resaltar su trabajado modo aprendizaje y su facilidad para diseñar diferentes IAs.

- Fritz: Si bien no tuvo tanta acogida por el público general (ni tampoco era el objetivo) sí que ha sido sin duda la plataforma más importante para usuarios avanzados. Desde el año 1995 en el que ganó el campeonato mundial de computadoras de ajedrez, Fritz fue creciendo hasta convertirse en el primer programa en obtener el título de maestro internacional. A lo largo de los años fue en cierta manera acoplándose a la moda y en 2009 hizo un cambio total de la interfaz así como una actualización de su base de datos, introduciendo 1.5 millones de partidas nuevas para contrastar jugadas. Como puntos positivos de su versión actual podemos mencionar su gran abanico de funciones, mucho mejor estructuradas que en anteriores entregas así como una interfaz visual más agradable. Ofrece una gran utilidad de auto-análisis así como la posibilidad de contrastar con bases de datos de todo el mundo una posición en el tablero. Aun así el sistema de entrenamiento no es tan completo como el de ChessMaster, además de que la IA a pesar de ser más inteligente/compleja no es tan fácil de modificar/diseñar como lo sería en ChessMaster (9).

- Backgammon: Otro de los grandes juegos de mesa que no solo pasó al modelo digital para ser jugado sino también para ser objeto de estudio. A partir de 1980 se empezaron a desarrollar inteligencias fuertes y se empezaron a crear programas para el público general.

  - Snowie: Es posiblemente el programa más importante para jugar a Backgammon, de cara a los usuarios y de cara a los jugadores profesionales. Cuenta con una interfaz depurada así como con una inteligencia artificial basada en un árbol de búsqueda propietario y un novedoso algoritmo de IA bastante más potente que los anteriores (10). Dentro de poco se podrá competir contra él por dinero. Posee un gran abanico de opciones enfocadas al desarrollo del jugador, como sus estadísticas y el análisis de movimientos post-partida.

  - GNU Backgammon: La alternativa a Snowie de código libre si bien no es tan visual como podría serlo Snowie, no podemos olvidar que es libre (gratis) y que cumple sus funciones de manera bastante correcta. Si bien no cuenta con algunas de las funciones más interesantes de Snowie como la base de datos/comparativa de jugadores, es cierto que cuenta con una IA bastante potente (11) hay que añadir que GNU es bastante más rápido en todas las áreas. Así como al ser código libe es bastante más adaptable a cualquier entorno (12).

- Colonos de Catán: El lanzamiento que creó escuela en la cultura moderna de los juegos de mesa está adaptado también al mundo digital.
    - Catán: El juego oficial nos brinda la posibilidad de jugar a los colonos de catán en casi cualquier plataforma conocida, tanto medios táctiles como videoconsolas u ordenadores. Si bien provee de la interfaz más bonita de todas las opciones posibles, es cierto que deja poca libertad a la hora de manipular al ser un producto de código cerrado. Señalar que la IA ha sido evolucionada desde su primer lanzamiento y que ahora cuenta con diferentes niveles, llegando a niveles de complejidad bastante altos en lo que se refiere a estrategias (13).

    - Settlers of Catan(distribuciones no-oficiales): Tras analizar varias distribuciones colgadas en internet, podemos llegar a la conclusión de que no son ni mucho menos visualmente tan bonitos como el juego oficial; pero que a la vez poseen muchas posibilidades al ser distribuidos compilados o directamente el código en Java. Alguno como **JSettlers2** da la posibilidad de jugar online con otros jugadores de catán de forma gratuita.

1.3.2.1.2 Clientes Web

Suelen ser bastante más sencillos gráficamente así como proveer de menos opciones al usuario, se centran mayoritariamente en jugar partidas estándar.

- Ajedrez: Cada vez hay más servidores Web que permiten jugar una partida interesante al usuario medio. Aun así sería necesario distinguir 2 tipos de portales de ajedrez, los basados en inteligencias sencillas que se pueden encontrar en cualquier web o algunos más trabajados como por ejemplo:
    - http://www.shredderchess.com/ : Este portal ofrece la posibilidad de jugar con una versión "recortada" del algoritmo Shredder, uno de los más importantes de finales de los 90 y que ha seguido participando en los campeonatos mundiales hasta la fecha.
- Backgammon: No existen muchos portales online de calidad con lo cual no nos extenderemos mucho en el análisis.
    - Yahoo Backgammon: De interfaz bastante amigable es un juego sencillo para jugar con una inteligencia artificial de carácter medio.
- Colonos de Catán:
    - PlayCatan: La misma compañía que ofrece Catán para medios físicos también ofrece la posibilidad de jugar online con otros jugadores o contra una IA. La interfaz está bastante trabajada, así como el sistema de cuentas de usuario. Destacar que ofrece la posibilidad de jugar a la versión básica o a las expansiones. Además de un tutorial bastante completo sobre cómo jugar.
    - JSettlers: Proyecto de código libre bien documentado que permite jugar a través de una interfaz web. Hay un applet montado en su servidor para poder jugar con otros jugadores o practicar contra la IA. Para jugar se necesita montar un servidor y acceder a través de un cliente web/explorador.

1.3.2.2 Investigación

Ya hemos analizado las aplicaciones o plataformas enfocadas a los usuarios y desde el punto de vista de un usuario. En este apartado haremos un análisis de los simuladores o plataformas de juegos de mesa digitalizados, que estén enfocados a los desarrolladores/investigadores, centrándonos en las opciones que ofrecen a la hora de trabajar, la complejidad de realizar los estudios así como los resultados que se pueden obtener.

- Ajedrez: En ajedrez una inteligencia no se define como tal, sino que se le llama motor de ajedrez. Para diseñar dichos motores existen varios protocolos estandarizados que serán los encargados de permitir a un motor comunicarse con una interfaz gráfica. (14)
    - Protocolos:
        - Winboard: Está basado en XBoard que fue el primer protocolo para Unix creado para comunicarse con una interfaz gráfica concreta, pero esto fue solo fue el comienzo ya que fue tomando importancia y se convirtió en un estándar. Winboard es la evolución de este estándar adaptado a Windows. A favor del Winboard se puede decir que es más elegante desde el punto de vista de un desarrollador, ya que pasa a la GUI únicamente la información necesaria, además de soportar un modo gráfico también puede correr una partida en modo consola. Debemos también destacar su la flexibilidad que proporciona a los desarrolladores para incluir nuevas funciones en los motores (15).
        - Universal Chess Protocol: Es un protocolo más moderno que el anterior diseñado especialmente para este cometido. Solo puede funcionar con interfaces gráficas, a través de las cuales ofrece no solo el estado actual de la partida si no también opciones de modificación de los motores. Además de su robustez, la interfaz gráfica muestra siempre exactamente que está haciendo el motor, así como que soporta bases de datos de finalización de partidas.

    - Interfaces:
        - Arena: Es una GUI gratuita que permite usar motores basados en cualquiera de los 2 protocolos anteriores. Si bien no está tan trabajado como los *GUIs* comerciales, ofrece un rendimiento general bastante bueno. Soporta el análisis de finalización de partidas Gaviota, además de información precisa de lo que hacen los motores en cada momento.
        - Fritz13: Esta interfaz ofrece unas características visuales inmejorables adoptando el estándar de office 2010. En esta versión se ha mejorado la gestión de bases de datos de partidas además del aumento de bases datos incluidas para contrastar partidas. Pero el elemento más importante de cara a los desarrolladores sería la función “Let’s check”, gracias a esta funcionalidad podremos analizar cada posición de tablero en conjunto con toda una red de jugadores mundial. Esto nos permitirá diseñar IAs que respondan a datos estadísticos más reales a la vez que de diferentes escenarios.

- Backgammon: No existe ninguna plataforma que realmente sea útil para investigadores. Destacar que de entre las IAs de renombre, la segunda más potente es de código abierto (GNUBackgammon) solo superada por ExtremeGammon que es un software de pago.

- Catan: Para este juego hemos analizado 2 versiones que consideramos pueden tener cierto interés desde el punto de vista del desarrollador/investigador.
    - Solitarie Settlers of Catan: Son seleccionables 4 tipos de inteligencia, que se gestiona a través de un enumerado, de manera que en la propia clase jugador el sistema reacciona usando un algoritmo diferente para cada método dependiendo de la inteligencia seleccionada. No es ni mucho menos un sistema fácilmente modulable ya que al albergar la gestión de IA dentro del funcionamiento normal es bastante complicado usar diferentes alternativas puesto que hay que modificar métodos específicos, que a la vez desarrollan más funciones que únicamente esperar una respuesta de una IA.

    - JSettlers2: Esta plataforma comprende tanto de un servidor así como de un cliente, con lo que está preparado. La parte que nos interesa que es el cliente, este tiene una arquitectura compleja en la que cuando un jugador es un robot, hace una gestión de mensajes y actúa de manera diferente ante cada mensaje. La estrategia está diseñada dentro de un método concreto para los temas generales y a la vez para algunas acciones más complejas tiene creadas clases externas. Esto a primera vista resulta algo complejo, ya que la misma clase robot tiene diferentes funciones aparte de la de gestión de mensajes y además aporta algunas clases extra para manejar situaciones específicas.

1.4 Dominion en la informática

El juego de Dominion creó al igual que Catán un antes y un después en los juegos de mesa. Si bien es cierto que no tuvo el mismo gran impacto social, no podemos pasar por alto que Dominion ha creado un nuevo nicho de juegos de mesa innovador, basado en la construcción de mazos. Al contar todos los jugadores con el mismo set inicial de cartas y al existir un azar casi nulo, únicamente al barajar cartas, nos encontramos con un estilo de juego que estará muy basado en la estadística a la hora de elaborar estrategias ganadoras.

En este apartado veremos las diferentes herramientas relacionadas con Dominion, desde simuladores, creadores de tableros hasta estudios estadísticos completos de partidas jugadas.

1.4.1 Isotropic

Ha sido la implementación online más famosa de Dominion. Mediante este cliente web, miles de jugadores pudieron probar sus cualidades y estrategias así como probar las expansiones del juego antes de comprarlas. Los puntos más destacables de este cliente web eran 2, su base de datos y su escala de jugadores.

- Base de datos: Al final de cada partida, toda la información de la misma se almacenaba en una base de datos para su posterior uso en análisis estadístico. Desde los ganadores de las partidas hasta que cartas se compraban, descartaban o ganaban en cada turno así como la media de puntos o tesoros que se tenían en cada turno.
- Escala de jugadores: La estimación de nivel de jugadores no era solo un punto clave de cara al emparejamiento de jugadores sino también a la hora de hacer análisis de la partida pudiendo agrupar partidas por el nivel de sus jugadores.

Con la salida del juego oficial se cerró el *website* de esta plataforma. Aun así la base de datos se dejó accesible para que esos datos no se perdieran y pudieran seguir siendo utilizados por otros proyectos o jugadores interesados.

1.4.2 Council Room

Este *site* contiene una interfaz de acceso a las partidas que se jugaron en el simulador de Isotropic. Gracias a esta interfaz es posible analizar 10.501.983 partidas jugadas y poder extraer información estadística de interés para los jugadores. El *site* ha evolucionado a lo largo del tiempo ofreciendo nuevos servicios para el uso avanzado de jugadores expertos, los cuales analizaremos en este apartado:

- Gráfico de porcentaje de victoria: En este gráfico analizamos el porcentaje de victorias al comprar una carta concreta, este información se muestra en un gráfico X/Y siendo Y el porcentaje de victoria y X el turno en el que se adquirió dicha carta.

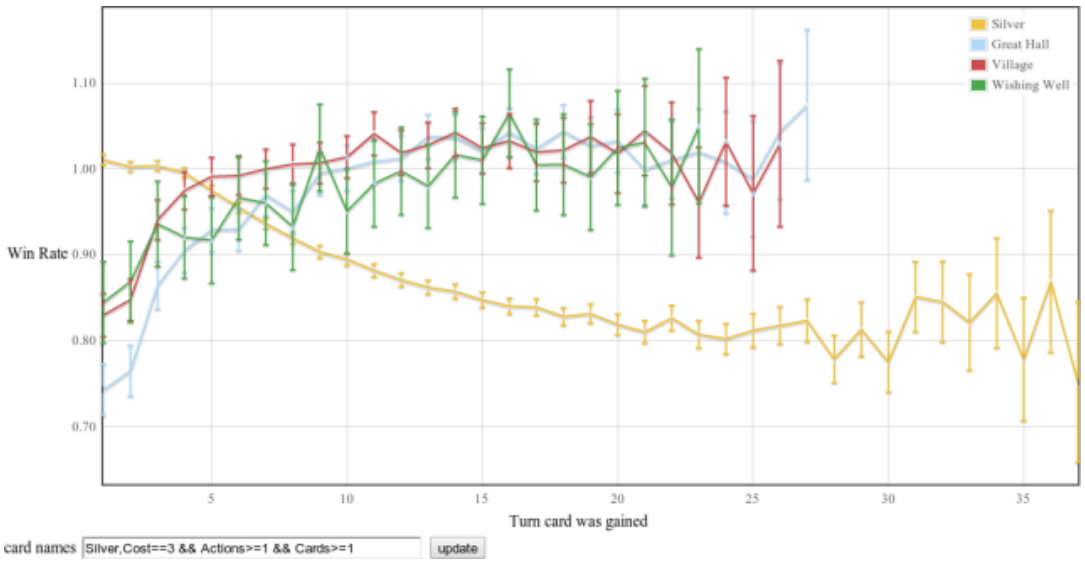


Ilustración 1. Gráfica de cartas/turno Council Room

Los datos de la ilustración 1 podrían ser accedidos por una IA para valorar qué carta adquirir en cada turno, pero esto sería erróneo porque si bien hay cartas que estadísticamente parecen más acertadas, no tienen por qué serlo en la realidad ya que aunque el índice de victoria es correlativo a la adquisición de la carta, la correlatividad no implica causalidad. Ganar una partida dependería de muchos más factores como las cartas que adquirieran los contrincantes así como el orden en el que se jueguen las cartas. Aun así estos indicadores pueden servir para seleccionar un grupo de cartas en cada turno sobre las cuales elegir la siguiente compra (16).

- Buscador de partidas: Definiendo los parámetros de nombres de jugador y las cartas de reino disponibles, la web nos ofrece un listado de los logs de las partidas jugadas que se adecuan a dichos parámetros.

Ilustración 2. Búsqueda de partidas

En los resultados que aparecen en la ilustración 2 podemos apreciar las cartas que adquirió el jugador en la partida así como las de todos los participantes, además de acceder al log completo de la partida mediante un click.

- Buscador de jugadores: Esta opción es bastante útil en caso de querer seguir la trayectoria de un jugador en concreto. Introduciendo el nombre de un jugador accederemos a un informe como el que aparece en la ilustración 3. Donde tendremos acceso a un estudio estadístico de todas sus partidas, sus records, su índice de victoria así como las expansiones con las que suele jugar dicho jugador.

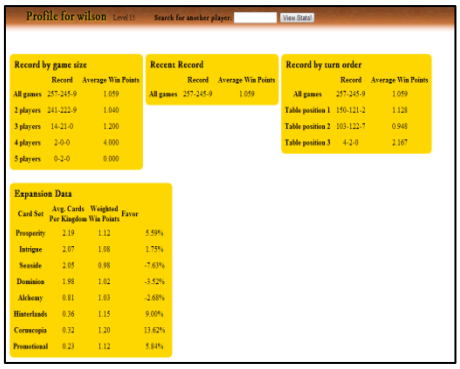


Ilustración 3. Estadísticas jugador

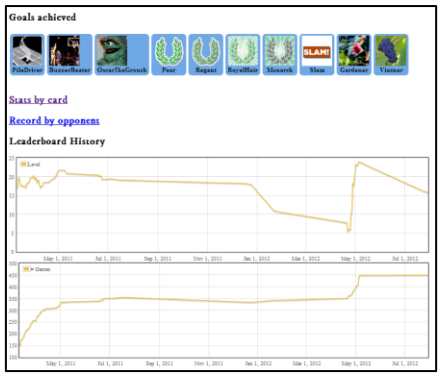
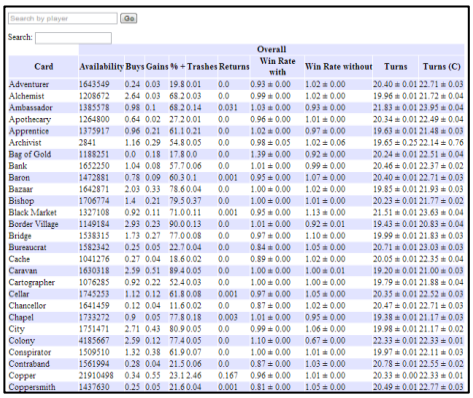


Ilustración 4. Histórico Jugador

También es posible encontrar un histórico de sus logros así como de su nivel como jugador a lo largo de su trayectoria en el servidor, como aparece en la ilustración 4. Desde esta página también podremos acceder a su histórico de oponentes y las estadísticas de cartas que ha usado.

- Compras populares: Quizás esta funcionalidad no es tan útil para el jugador, pero sin duda es interesante saber qué carta es la más usada por los jugadores y cuáles son los desenlaces de las partidas en las cuales aparece. En la ilustración 5 podemos ver como se muestra esta información de compras populares.



Search by player: Go

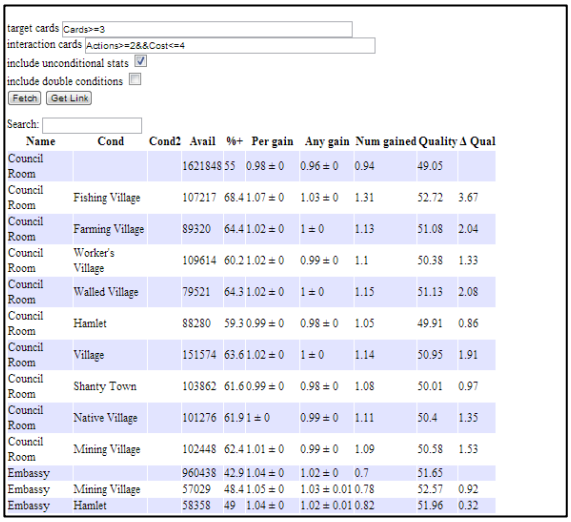
Search:

Card	Availability	Buys	Gains	% +	Trashes	Returns	Overall Win Rate with	Win Rate without	Turns	Turns (C)
Adventurer	1643549	0.24	0.03	19.8	0.01	0.0	0.93 ± 0.00	1.02 ± 0.00	20.40 ± 0.01	22.71 ± 0.03
Alchemist	1208672	2.64	0.03	68.2	0.03	0.0	0.99 ± 0.00	1.02 ± 0.00	19.96 ± 0.01	21.72 ± 0.04
Ambassador	1385578	0.98	0.1	68.2	0.14	0.031	1.03 ± 0.00	0.93 ± 0.00	21.83 ± 0.01	23.95 ± 0.04
Apothecary	1264800	0.64	0.02	27.2	0.01	0.0	0.96 ± 0.00	1.01 ± 0.00	20.34 ± 0.01	22.49 ± 0.04
Apprentice	1373917	0.96	0.21	61.1	0.21	0.0	1.02 ± 0.00	0.97 ± 0.00	19.63 ± 0.01	21.48 ± 0.03
Archivist	2841	1.16	0.29	54.8	0.05	0.0	0.98 ± 0.05	1.02 ± 0.06	19.65 ± 0.25	22.14 ± 0.76
Bag of Gold	1188251	0.0	0.18	17.8	0.0	0.0	1.39 ± 0.00	0.92 ± 0.00	20.24 ± 0.01	22.51 ± 0.04
Bank	1652250	1.04	0.08	57.7	0.06	0.0	1.01 ± 0.00	0.99 ± 0.00	20.46 ± 0.01	22.37 ± 0.02
Baron	1472881	0.78	0.09	60.3	0.1	0.001	0.95 ± 0.00	1.07 ± 0.00	20.40 ± 0.01	22.71 ± 0.03
Bazaar	1642871	2.03	0.33	78.6	0.04	0.0	1.00 ± 0.00	1.02 ± 0.00	19.85 ± 0.01	21.93 ± 0.03
Bishop	1706774	1.4	0.21	79.5	0.37	0.0	1.00 ± 0.00	1.01 ± 0.00	20.23 ± 0.01	21.77 ± 0.02
Black Market	1327108	0.92	0.11	71.0	0.11	0.001	0.95 ± 0.00	1.13 ± 0.00	21.51 ± 0.01	23.63 ± 0.04
Border Village	1149184	2.93	0.23	90.0	0.13	0.0	1.01 ± 0.00	0.92 ± 0.01	19.43 ± 0.01	20.83 ± 0.04
Bridge	1538315	1.73	0.27	77.0	0.08	0.0	0.97 ± 0.00	1.10 ± 0.00	19.99 ± 0.01	21.83 ± 0.03
Bureaucrat	1582342	0.25	0.05	22.7	0.04	0.0	0.84 ± 0.00	1.05 ± 0.00	20.71 ± 0.01	23.03 ± 0.03
Cache	1041276	0.27	0.04	18.6	0.02	0.0	0.89 ± 0.00	1.02 ± 0.00	20.05 ± 0.01	22.35 ± 0.04
Caravan	1630318	2.59	0.51	89.4	0.05	0.0	1.00 ± 0.00	1.00 ± 0.01	19.20 ± 0.01	21.00 ± 0.03
Cartographer	1076285	0.92	0.22	52.4	0.03	0.0	1.00 ± 0.00	1.00 ± 0.00	19.79 ± 0.01	21.88 ± 0.04
Cellar	1745253	1.12	0.12	61.8	0.08	0.001	0.97 ± 0.00	1.05 ± 0.00	20.35 ± 0.01	22.52 ± 0.03
Chancellor	1641459	0.12	0.04	11.6	0.02	0.0	0.87 ± 0.00	1.02 ± 0.00	20.47 ± 0.01	22.71 ± 0.03
Chapel	1733272	0.9	0.05	77.8	0.18	0.003	1.01 ± 0.00	0.95 ± 0.00	19.38 ± 0.01	21.17 ± 0.03
City	1751471	2.71	0.43	80.9	0.05	0.0	0.99 ± 0.00	1.06 ± 0.00	19.98 ± 0.01	21.17 ± 0.02
Colony	4185667	2.59	0.12	77.4	0.05	0.0	1.10 ± 0.00	0.67 ± 0.00	22.33 ± 0.01	22.33 ± 0.01
Conspirator	1509510	1.32	0.38	61.9	0.07	0.0	1.00 ± 0.00	1.01 ± 0.00	19.97 ± 0.01	22.11 ± 0.03
Contraband	1561994	0.28	0.04	21.5	0.06	0.0	0.87 ± 0.00	1.03 ± 0.00	20.78 ± 0.01	22.55 ± 0.02
Copper	21910498	0.34	0.55	23.1	2.46	0.167	0.96 ± 0.00	1.01 ± 0.00	20.33 ± 0.00	22.33 ± 0.01
Coppersmith	1437630	0.25	0.05	21.6	0.04	0.001	0.81 ± 0.00	1.05 ± 0.00	20.49 ± 0.01	22.77 ± 0.03

Ilustración 5. Comprar populares

Además tienes la posibilidad de filtrar los resultados para obtener los datos relativos a las cartas más compradas por un jugador concreto.

- Estadísticas de correlatividad de cartas: Este servicio de la web nos ofrece una interfaz de estadísticas relacionadas con lo útiles que son unas cartas u otras dependiendo de la existencia de otras cartas en el mazo o en los suministros. En los parámetros que podemos usar para filtrar la información podemos incluir la carta objetivo así como acotar el grupo de cartas con el que queremos relacionar. En la ilustración 6 podemos ver un ejemplo de dichas estadísticas.



target cards: Cards>=3

interaction cards: Actions>=2&&Cost<=4

include unconditional stats: ☑

include double conditions: ☐

Fetch | Get Link

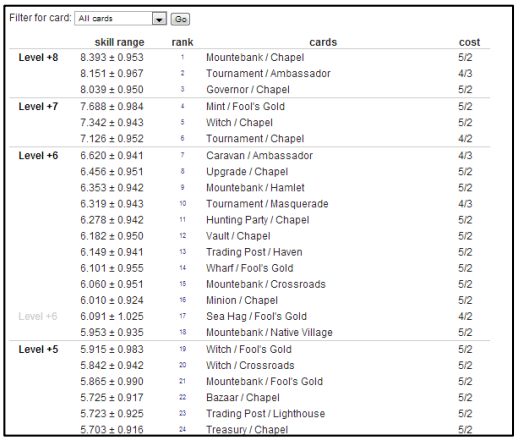
Search:

Name	Cond	Cond2	Avail	%+	Per gain	Any gain	Num gained	Quality	Δ Qual
Council Room			1621848	55	0.98 ± 0	0.96 ± 0	0.94	49.05	
Council Room	Fishing Village		107217	68.4	1.07 ± 0	1.03 ± 0	1.31	52.72	3.67
Council Room	Farming Village		89320	64.4	1.02 ± 0	1 ± 0	1.13	51.08	2.04
Council Room	Worker's Village		109614	60.2	1.02 ± 0	0.99 ± 0	1.1	50.38	1.33
Council Room	Walled Village		79521	64.3	1.02 ± 0	1 ± 0	1.15	51.13	2.08
Council Room	Hamlet		88280	59.3	0.99 ± 0	0.98 ± 0	1.05	49.91	0.86
Council Room	Village		151574	63.6	1.02 ± 0	1 ± 0	1.14	50.95	1.91
Council Room	Shanty Town		103862	61.6	0.99 ± 0	0.98 ± 0	1.08	50.01	0.97
Council Room	Native Village		101276	61.9	1 ± 0	0.99 ± 0	1.11	50.4	1.35
Council Room	Mining Village		102448	62.4	1.01 ± 0	0.99 ± 0	1.09	50.58	1.53
Embassy			960438	42.9	1.04 ± 0	1.02 ± 0	0.7	51.65	
Embassy	Mining Village		57029	48.4	1.05 ± 0	1.03 ± 0.01	0.78	52.57	0.92
Embassy	Hamlet		58358	49	1.04 ± 0	1.02 ± 0.01	0.82	51.96	0.32

Ilustración 6. Gráfico de correlatividad entre cartas y victorias

Esta estadística si resulta bastante útil, ya que podría usarse en un algoritmo de valoración de cartas que formara parte de una IA de Dominion.

- Mejores/peores aperturas: El *site* define apertura a las 2 primeras cartas compradas, como podemos ver en la ilustración 7. Además considera que las mejores aperturas están acotadas según niveles, es decir, si una apertura gana siempre pero es solo con jugadores de cierto nivel, quedará reducida a dicho alto nivel. Por el contrario, puede ser que otra apertura en un nivel 7/8 no llegara a funcionar pero que a niveles bajos fuera la más efectiva.



Filter for card: All cards Go

	skill range	rank	cards	cost
Level +8	8.393 ± 0.953	1	Mountebank / Chapel	5/2
	8.151 ± 0.967	2	Tournament / Ambassador	4/3
	8.039 ± 0.950	3	Governor / Chapel	5/2
Level +7	7.688 ± 0.984	4	Mint / Fool's Gold	5/2
	7.342 ± 0.943	5	Witch / Chapel	5/2
	7.126 ± 0.952	6	Tournament / Chapel	4/2
Level +6	6.620 ± 0.941	7	Caravan / Ambassador	4/3
	6.456 ± 0.951	8	Upgrade / Chapel	5/2
	6.353 ± 0.942	9	Mountebank / Hamlet	5/2
	6.319 ± 0.943	10	Tournament / Masquerade	4/3
	6.278 ± 0.942	11	Hunting Party / Chapel	5/2
	6.182 ± 0.950	12	Vault / Chapel	5/2
	6.149 ± 0.941	13	Trading Post / Haven	5/2
	6.101 ± 0.955	14	Wharf / Fool's Gold	5/2
	6.060 ± 0.951	15	Mountebank / Crossroads	5/2
	6.010 ± 0.924	16	Minion / Chapel	5/2
Level +6	6.091 ± 1.025	17	Sea Hag / Fool's Gold	4/2
	5.953 ± 0.935	18	Mountebank / Native Village	5/2
Level +5	5.915 ± 0.983	19	Witch / Fool's Gold	5/2
	5.842 ± 0.942	20	Witch / Crossroads	5/2
	5.865 ± 0.990	21	Mountebank / Fool's Gold	5/2
	5.725 ± 0.917	22	Bazaar / Chapel	5/2
	5.723 ± 0.925	23	Trading Post / Lighthouse	5/2
	5.703 ± 0.916	24	Treasury / Chapel	5/2

Ilustración 7. Estadísticas de jugadas de apertura

- Gracias a esta organización de la información podemos valorar dichas aperturas de una manera mucho más apropiada a la vez que útil si queremos usar estos datos para hacer estudios pormenorizados a la hora de crear estrategias de juego.

1.4.3 Dominion Card Randomizer

Esto no es nada más y nada menos que un creador de tableros, atendiendo a una serie de parámetros podemos acotar sobre que cartas iterará el simulador para crear el tablero. Su característica principal es añadir o quitar expansiones, con lo cual aumentaremos o disminuiremos el número de cartas del reino. Además de esta función principal, podemos definir varías opciones específicas, como incluir el foso en caso de que haya alguna carta de ataque, incluir cartas con efectos específicos como sumar acciones/tesoros y por último, una carta que permita eliminar cartas de tu mano. En la ilustración 8 podemos ver la interfaz de opciones.

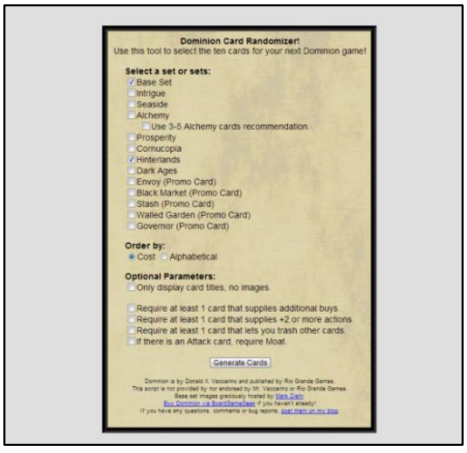


Ilustración 8. Menú de Card Randomizer

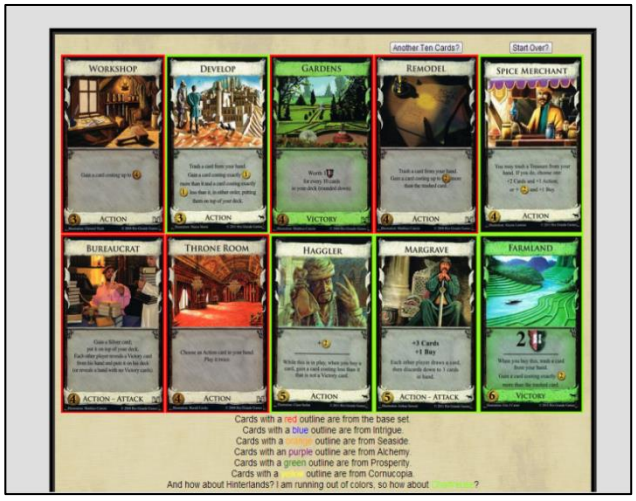


Ilustración 9. Resultado en modo gráfico

Además, podemos seleccionar si queremos que nos muestre las cartas elegidas con imágenes, como podemos ver en la ilustración 9, o en texto plano. Al mostrar las cartas además indicará a qué expansión pertenecen en caso de que no fueran del juego básico, que también lo indicaría. En caso de que no nos guste la selección podremos repetir para obtener 10 cartas diferentes que correspondan a los mismos parámetros o en caso de que quisiéramos cambiar completamente, también existe la posibilidad de volver a la interfaz inicial.

1.4.4 Dominion Strategy

Este blog de estrategia hecho por y para jugadores de Dominion contiene una gran amplitud de artículos, relacionados con estrategias de juego, desde análisis de cartas, combos y contraataques a partidas anotadas. Incluyendo una Wiki, un glosario así como un canal propio de YouTube. En la ilustración 10 se puede ver su página principal.

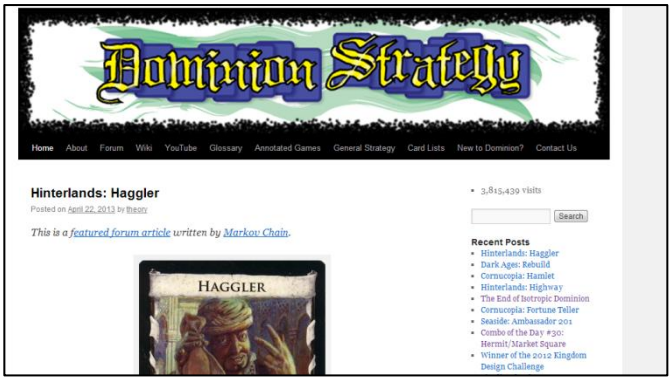


Ilustración 10. Dominion Strategy FrontPage

En el foro, que vemos en la ilustración 11, podemos encontrar diferentes subforos bastante interesantes que abarcan temas como simuladores, torneos o reglas así como un subforo personal para el creador de Dominion (*Donald X.).* Además, hay interesantes propuestas de mini-juegos/retos como vaciar x pilas antes del turno 4 y otros retos similares que pongan en juego la astucia del jugador. En el mismo foro podremos encontrar logs de partidas jugadas así como un subforo especial dedicado a fans en donde podrán exponer sus ideas o creaciones para Dominion.

Ilustración 11. Foro de Dominion Strategy

1.4.5 Dominiate

Dominiate es el primer simulador de los dos que analizaremos en el estudio del estado del arte. Su funcionamiento es sencillo comparado con el simulador de Geronimoo. Aunque tanto el uno como el otro, únicamente ofrecen la posibilidad de enfrentar unas IAs con otras excluyendo la posibilidad de jugar como usuario. El simulador ofrece dos interfaces, una gráfica ejecutable a través de un navegador web y una estándar por consola.

- Funcionalidades: Aquí describiremos brevemente qué ofrece este simulador.
  - **Jugadores/inteligencias:** De 2 a 4, dependiendo de la interfaz de usuario que se use.
  - **Inteligencia Artificial:** Este simulador da la posibilidad de poder evaluar de manera independiente cada carta, además de dar la posibilidad de priorizar una lista de cartas para las diferentes acciones del juego (jugar acción, descartar carta, ganar carta, etc…)
  - **Análisis:** Al final de la simulación es posible ver un gráfico de análisis de puntos de victoria y dinero medio por turno así como el historial de acciones realizadas por los jugadores de cualquiera de las partidas simuladas.
- Diseño: Es un diseño sencillo comparado con un simulador de juegos más completo, como Geronimoo o el JSettlers, esto se debe principalmente a que no está pensado para ser ampliado o dar una gran cobertura de uso. Está implícito que un simulador basado en un lenguaje de scripting y por lo tanto, que tendrá que ser ejecutado a través de un navegador web, no será muy pesado, sea ejecutado remotamente o de forma local.
  A la hora de agrupar funcionalidades o de definir el funcionamiento el programador ha definido las siguientes clases que definiremos una por una.

  - **playWeb:** Este script es el que posibilita usar el simulador a través de una interfaz web, las opciones de uso se limitan a las que ofrece la interfaz gráfica limitándolo a dos bots corriendo a la vez.
  - **Play:** Este script posibilita correr el simulador en modo consola, a la vez que hace posible el desarrollo de la partida por línea de comandos. Resaltar que corriendo el simulador en modo consola es posible hacer funcionar más de dos bots de manera simultánea.

- **GameState:** Este módulo/script define desde los suministros, el estado de la partida hasta el propio funcionamiento del juego. Define dos clases básicas que se encargaran de contener toda la información y métodos de funcionamiento. Una clase está relacionada con el jugador y con la IA que toma decisiones en su nombre y otra clase que almacena el resto de la información, así como define el funcionamiento de Dominion y además contiene las funciones extra necesarias para el correcto desarrollo del juego. State será la clase principal que interconecta los diferentes módulos de juego. Así como la clase PlayerState solo contiene información relativa al jugador que será la modificada por las cartas de acción y tesoro. Cada vez que una IA quiera jugar una carta o tomar una decisión, obtendrá la información del estado de su jugador y sobre esta información tomará la decisión correspondiente y le devolverá la decisión a la clase State que será la encargada de resolver las acciones y efectos.

- **basicAI:** Esta clase es la que es consultada para todas las decisiones del juego. Cuando se entra en una fase del juego (Acción, Compra, Mantenimiento) la clase State que es la que alberga en funcionamiento del juego, llama al jugador correspondiente y le pasa por parámetro las opciones sobre las que iterar así como el contexto de dicha elección, ya sea llamando a una función específica, o pasando un String por parámetro a una función de consulta general. Estas funciones abarcan la gran mayoría de decisiones, pero además el programador ha incluido funciones específicas para tomar decisiones que no responden a efectos generales. Estas funciones son normalmente invocadas desde las cartas que llevan su nombre, como por ejemplo "*embajador*". Dichas opciones sobre las que iterar son las diferentes cartas que podremos elegir en ese momento determinado de la partida. A la hora de decidir qué carta ganar/comprar/descartar podemos definir una lista de prioridades que la IA tendrá siempre en cuenta. El juego viene con ciertas listas de prioridades de adquisición/compra/descarte y similares definidas de antemano, pero además el navegador Web nos da la posibilidad de crear las nuestras propias y utilizarlas. Aparte de la lista de prioridades ya descrita la IA dispone de otro camino para tomar las decisiones, que es el valor de la cartas, mediante una serie de funciones matemáticas predefinidas se les otorga un valor general a cada carta a la hora de priorizar para descartarlas, eliminarlas, obtenerlas, etc… además a la hora de jugar una carta se invoca una función de la propia carta.

- **cards:** En este script el programador ha definido todas las cartas posibles del juego. A través de un grupo básico de cartas crea un abanico de posibilidades que usarán otras cartas más complejas mediante la herencia para dar cabida al máximo número posible de cartas de Dominion. Cuando se crea una de estas cartas complejas es posible re-definir el método que se ejecuta al jugar la carta, pudiendo crear cartas con efectos totalmente personalizables o que no atienden a la estructura creada por las cartas básicas. Además, en este objeto está definida la función playValue() que será invocada a la hora de jugar un carta para evaluar la mejor opción de todas las cartas posibles, cuanto más alto sea el valor mejor.

Tras analizar todos los scripts/módulos que hacen posible el funcionamiento del simulador haremos un breve resumen con conclusiones relativas al diseño del mismo.

- **Lenguaje:** El uso de coffee script como lenguaje nativo del juego es una decisión un poco arriesgada y quizás no muy útil. Desde mi punto de vista, siendo este un simulador de código libre que está a la espera de ser mejorado y expandido por la red por una serie de desarrolladores desconocida, hubiese sido mejor elección usar JavaScript directamente. Y digo esto porque aparentemente los beneficios de coffee script son meramente visuales de cara a la sintaxis, con lo cual, creo que la decisión acertada hubiera sido JavaScript. Aun así, reconocer la acertada elección de usar el lenguaje de scripting como medio para implementar el simulador (y digo acertada porque como bien dice el autor) no plantea nunca una alternativa al simulador Geronimoo en lo que a potencia/funciones se refiere, con lo cual crear una versión ligera de fácil instalación/uso es desde luego un acierto.
  Esto lo decimos asumiendo también la dificultad de una posible ampliación/expansión debido a que en este caso ligereza supone aglutinar muchas funcionalidades en las mismas clases/scripts haciendo más complicado el trabajo de futuros desarrolladores.

- **Organización:** El diseño simplista ayuda por supuesto a la implementación, pero quizás no tanto a la hora posibilitar a otros desarrolladores a mejorarlo o ampliarlo. Podrían haberse usado diferentes scripts para encapsular los diferentes métodos o clases, separando de una manera más concisa el jugador del tablero. Así como crear también un script vacío o básico que definiera estrategias para todo y poder heredar de ese a la hora de definir estrategias nuevas para cada opción. Lo que hemos visto es un script de IA que engloba todas las funciones de decisión pero que realmente da pocas posibilidades de modificación. No solo debido a la complejidad del mismo si no que habría que retocar toda la clase para probar los nuevos métodos. Al usuario medio se le brinda la posibilidad de definir una lista de prioridades a la hora de definir qué cartas debe adquirir/ganar la IA pero ahí se acabaron las posibilidades de modificación, "*sencillas*".

- **Definición de cartas:** Si bien se han tratado de manera inteligente intentando agrupar ciertas cartas básicas como objetos y haciendo heredar a las demás de ese grupo de cartas básicas, quizás el formato es algo rígido, cada vez que se quiera crear una carta nueva habrá que crear un método nuevo completo a la vez que a la hora de crear efectos complejos tendremos que rescribir el mismo código una y otra vez para cartas que difieran simplemente en detalles. Habría que buscar una manera de poder reutilizar código a la vez que definir las cartas en unidades incluso más pequeñas que la propia carta, atendiendo a los efectos que producen en el juego cada una.

- **Implementación:** A la hora de analizar la implementación de cada uno de los módulos del simulador intentaré hacer bastante hincapié en el orden secuencial en el que se van ejecutando los comandos para poder compararlo con el funcionamiento de mi propio proyecto. Además de cómo y dónde almacena las cartas, los suministros, como se modifica el estado de la partida, funciones más relevantes, etc…

- **playWeb:** Está definida para ser el medio por el que podamos hacer competir 2 inteligencias a través de un servidor web. Las funciones básicas que define este script, ya que es código incrustado, son invocadas desde la página HTML que muestra el navegador.
  - **CompileStrategies():** Esta función cargará en la variable de retorno strategies, las estrategias que le han pasado por parámetro una vez comprobadas que realmente existen, haciendo un "*matching*" entre los nombres introducidos por parámetro y los ficheros de estrategias.
  - **MakeStrategy():** Se le invoca al llamar a playGame() y será la función que inicialice las estrategias que se han compilado anteriormente y que se le han pasado por parámetro.
  - **playGame():** Esta función inicializa las IAs, así como el estado inicial de la partida dependiendo de las opciones elegidas por el usuario. Una vez inicializado el estado, se invoca la función DoPlay() de la clase State que inicializará el juego. Mientras tanto este script continuará funcionando gestionando los errores así como el gráfico de histórico de los jugadores a través de *.grapher.*

- **Play:** Este módulo es el que se encarga de hacer funcionar el simulador en caso de que queramos cargarlo a través de la consola de comandos y no a través de la interfaz web. Este script funciona de manera un poco diferente ya que tiene que importar módulos mediante la función *require()*. Además de esto las funciones funcionan de manera algo diferente ya que ahora el compilado y la carga de las IAs no son funciones independientes y funcionan de manera conjunta.

  Al no ser invocado a través de una interfaz web, este script cuenta con una función main de arranque para inicializar el simulador.

- **loadStrategy:** Recibe el nombre del fichero de estrategias por parámetro, crea un nuevo objeto IA, carga el fichero en memoria e introduce en el objeto IA la lista de prioridades del script de la estrategia.
- **playGame:** Recibe por parámetro un conjunto de nombres que determinan las estrategias a seguir de las IAs así como cuantos jugadores habrá, de los cuales cada uno tendrá una IA asignada. Después se inicializa el estado del objeto State con los jugadores y unas opciones predefinidas, al contrario que a través de la interfazWeb donde eran totalmente dinámicas. Mientras la partida no se acabe esta función imprimirá en el fichero log, el resultado de cada turno.

- **GameState:** Este script almacena las 2 clases que definen la partida y que analizaremos por separado.
  - **PlayerState:** Esta clase almacena en variables toda la información relativa al estado del jugador, que será la información que tome en consecuencia la IA a la hora de decidir acciones.
    - **Initialize:** Este método inicializa el jugador antes de empezar la partida y obtiene por parámetro la IA del mismo. El jugador posee 2 variables que almacenan el estado de la carta que jugamos y la que ganamos para no actuar sobre la misma carta 2 veces o de manera errónea. Además contiene una pila de acciones que contendrá las acciones en juego aun no resueltas.
    - **Métodos informativos:** el programador ha implementado una serie de funciones que provean de información a la IA a la hora de tomar decisiones; como las cartas que tiene en mano, las que tiene en el mazo, cuantas tiene de un tipo, etc...
    - **Métodos de modificación:** Dos métodos que interactúan con la clase jugador. Uno obtiene cartas del mazo y el otro permite obtener cartas del mazo hasta encontrar una concreta y qué hacer con la misma.
- **basicAI:** Esta clase crea los objetos que tomarán las decisiones que cada jugador efectúe en la partida. Los métodos implementados están divididos en diferentes grupos por el programador, nosotros seguiremos la misma organización a la hora de analizarlo.
  - **Maquinaria de decisión:** Devuelve la opción preferida de la AI. Primeramente obtiene el estado actual del jugador al que representa a través de la función myPlayer(). Una vez almacenado este estado en el algoritmo, busca en la lista de prioridades explicitas si alguna de ellas encaja en la lista de opciones posibles. En caso de que no, el algoritmo devolverá la opción valorando las opciones de manera singular a través de la función getChoiceValue().

- **Estrategias por defecto:** A la hora de obtener/comprar una carta seguimos la estrategia de prioridades mediante la función gainPriority(). En caso de que no exista prioridad, este método da un valor a cualquier carta mediante una función matemática estándar.
    - **Prioridades:** Esta IA está diseñada para que cada acción del tipo jugar tesoros o descartar cartas tenga una lista de prioridades de elección y en caso de que no, se llame a una función que evalúe las opciones de manera individual, dando un valor numérico a cada una. Hay cartas que por sus singularidades y el diseño del juego, necesitan que se analicen de una manera específica a la hora de valorar las acciones que se toman cuando se juegan.
    - **Decisiones complejas:** En algunos momentos dentro de una acción compleja, como cuando enviar una carta a la pila de trash no es una operación trivial, se llama a una función específica que está dentro de esta misma clase que nos devolverá la opción óptima a la hora de mandar a la pila de trash una carta.
    - **Métodos informativos**: Este grupo de métodos nos informan de qué cartas podríamos descartar o eliminar así como de cuantos puntos de victoria tenemos por el momento.

- **cards:** Todas las cartas del juego y sus correspondientes efectos están definidas en este archivo. Cada carta creada será inmutable y siguen el patrón di diseño Singleton. Todas las cartas creadas estarán contenidas en la variable c y podrán ser accedidas por su nombre. Las cartas se crean al cargar el modulo mediante llamadas continuas al método MakeCard(), el cual creará un objeto carta siguiendo las indicaciones que se pasan por parámetro.

    - **basicCard:** Este objeto define la estructura de las cartas del juego, sus atributos y los métodos de acceso a dichos atributos. Así como el valor de la carta de cara a la IA. Define métodos bastante complejos haciendo que la máquina pueda reaccionar sobre cualquier acción que se realice sobre la carta, desde jugarla, descartarla, ganarla o comprarla hasta barajarla en el mazo.
        - **MakeCard(name, origCard, props, fake):** Mediante este método crearemos todas las cartas del juego mediante pseudoherencia. El funcionamiento es crear una variable newCard que almacene las mismas claves que tenía almacenada la carta en la que está basada, que se pasa mediante el parámetro origCard. Las propiedades que difieren de la carta original se pasan por parámetro y se almacenan en la misma matriz de claves, ya sean modificaciones de los atributos o de las funciones que componen la clase.

- **Métodos de utilidades:** Estos métodos definen algunas funciones para transferir cartas entre listas. Así como de funcionamiento, como para aplicar efectos típicos de las carta.
    - **upgradingChoices():** Se usa cuando el proceso de elección de carta se renueva tras el uso de una acción por parte del jugador. Ambos métodos de funcionamiento reciben por parámetro el estado del juego para poder actuar sobre el estado real de la partida o en consecuencia de este.
- **Exports:** El módulo abre el uso externo de las funciones de gestión de listas mediante la función *export*, esto se hace porque estas funciones podrán ser invocadas desde otros scripts.

- **Interfaz:** Este simulador dispone de dos interfaces. Una web a través de un archivo HTML y otra que funciona a través de una consola de comandos mucho más sencilla que no analizaremos, ya que lo que necesitamos es analizar las funcionalidades generales.
    - **InterfazWeb:** Una sencilla interfaz con 2 ventanas para seleccionar/modificar estrategias. Un cuadro de opciones y una consola donde aparecerá el log de la partida y unas ventanas de gráficos. Podemos verla en la ilustración 12.

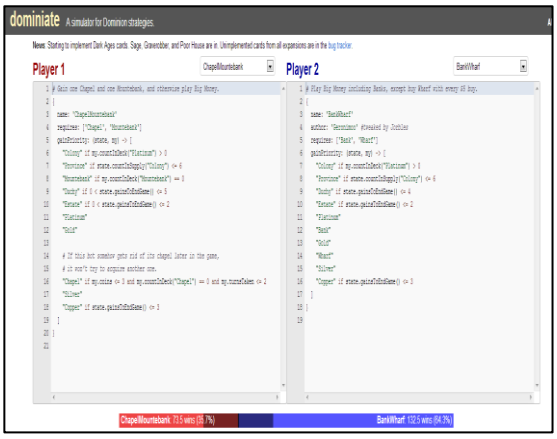


Ilustración 12. Ventana de estrategia

- **Ventanas de estrategia:** Te permite sobrescribir una serie de funciones de prioridad que el jugador debe conocer de antemano. Se ofrece la posibilidad de seleccionar una de las estrategias predefinidas y modificarla como se quiera; como se puede ver en la ilustración 1.
- **Cuadro de opciones:** Entre las opciones que el simulador brinda al usuario está la posibilidad de jugar 100 o 1000 partidas con las mismas estrategias, jugar 1 completa o solo hasta que haya un claro ganador. Además da la opción de ordenar al azar qué jugador jugará primero en cada partida y por último, incluye las cartas de "Colonia" y "Platino".
- **Consola de log:** Aparecen los movimientos de cada jugador durante la partida que seleccionemos de todas las jugadas; así como una barra estadística donde se recuenta el total de partidas ganadas por cada jugador con un color para cada uno.
- **Ventanas de gráficos:** Aparecen dos gráficos XY. Uno para los puntos de victoria y otro para las monedas, donde la variable Y representa puntos o monedas respectivamente y la variable X el turno de la partida. Podemos ver los gráficos en la ilustración 13.

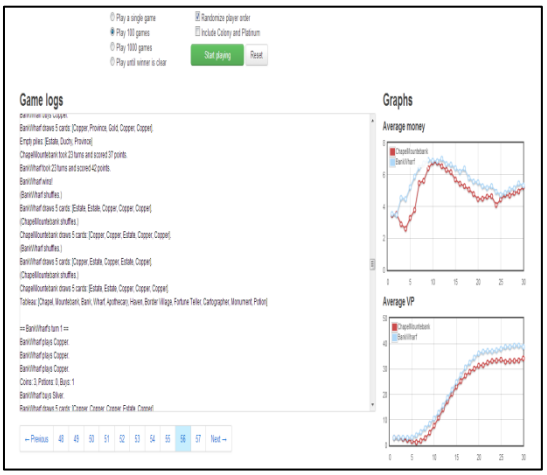


Ilustración 13. Ventana de Gráficos

- **Documentación:** No hay documentación externa ninguna sobre el funcionamiento del simulador. Todo el código está comentado en los mismos ficheros y mostrado de una manera bastante elegante gracias a Docco, un generador de documentación hecho en Coffeescript que genera documentos HTML, con los comentarios en la mitad izquierda de la pantalla y el código en la mitad derecha. Si bien faltan algunos métodos por explicar así como parte del funcionamiento del programa, la parte comentada/documentada está hecha la claridad y extensión suficientes.

1.4.6 Geronimoo's Simulator

Este simulador de Dominion es el más potente que ahora mismo se puede encontrar en la red. Está programado en Java y es totalmente autónomo. Es una herramienta de gran utilidad que permite lanzar miles de partidas en pocos segundos con hasta 4 jugadores jugando de manera simultánea y que a la vez permite analizar los resultados de dichas partidas de forma eficaz. La IA se compone de dos partes diferenciadas, una relacionada con la fase de compra y otra con la de jugar acciones que más adelante explicaremos con más profundidad.

Siendo este simulador la referencia más importante del estado del arte, el análisis del mismo será más profundo y detallista con el objetivo de poder entender a la perfección el funcionamiento; sacando a la luz sus puntos fuertes y debilidades para poder utilizar este conocimiento en nuestro propio proyecto.

- **Funcionalidades:**
  - Es capaz de similar 1000 partidas en segundos.
  - Acepta hasta 4 jugadores.
  - Tiene implementados todas las ampliaciones de Dominion.
  - Se pueden editar las cartas en la mano, mazo y pila de descartes iniciales del jugador.
  - Se pueden forzar las siguientes 2 configuraciones de inicio $4/PV3 o $5/PV2.
  - Mantener el orden de los jugadores.
  - Gráfico del punto de victorias medio en cada turno por cada jugador.
  - Una barra estadística que representan las victorias de cada jugador y los empates.
  - Más de 50 estrategias prediseñadas.
  - Interfaz fácil e intuitiva para crear y modificar estrategias de IA.
  - Bots que actúan siguiendo unas reglas de compra (IA limitada).

- **Organización:** En esta parte hablaremos del diseño del sistema visto desde arriba sin entrar en excesivos detalles de implementación que irán en el siguiente punto del análisis. El sistema está dividido en 4 paquetes y se ha usado una clase para cada elemento del juego físico así como para cada parte que compone la IA y paquetes específicos para los enumerados, las cartas y otro para la Interfaz Gráfica.
  - **dominionSimulator:** Este paquete base engloba a todas las clases básicas de funcionamiento a la vez que funciona como paquete raíz para el resto de paquetes. Para entender mejor el contenido de este paquete analizaremos una por una sus clases más características así como las acciones más representativas del juego.
    - **DomGame:** Esta será la encargada de coordinar la totalidad de las partidas, desde inicializar el tablero y jugadores hasta gestionar los turnos de cada uno de ellos. Siendo esta la única clase que tiene referencia al tablero, será la encargada de gestionar la comunicación entre el resto de clases y la clase tablero. Con esto se consigue que los jugadores no hagan "trampas" y puedan saltarse los medios que ofrece la clase Game para interactuar con el tablero. De hecho, muchas de las funciones implementadas en esta clase es únicamente son únicamente responsables de hacer la invocación de otra función con la misma nomenclatura implementada en la clase tablero.

Además de los jugadores, los métodos implementados en las clases relacionadas con las cartas, sus reglas de adquisición y compra también llamarán a la clase DomGame en vez de al tablero.

- **DomBoard:** Es la clase que define el tablero del juego y por lo tanto gestionará todas las acciones relacionadas con el manejo de cartas. En este manejo de cartas incluimos la gestión de pilas de provisión así como la pila de cartas eliminadas, las excluidas, los premios y todas las funciones de información que pudieran estar relacionadas con dichas pilas. De la gestión de las cartas cabe señalar que una vez son creadas las instanciaciones al inicializar el tablero, dichos objetos no desaparecerán en toda la simulación. Si no que a la hora de resetear la partida para seguir con la simulación, las cartas serán reasignadas a sus pilas de origen.

- **DomPlayer:** Esta clase representará al jugador en el simulador. Es por ello que en su diseño se ha tenido en cuenta toda la información que pertenece intrínsecamente al jugador y se han definido tanto atributos, información como su nombre, victorias, derrotas y empates, su mazo, las cartas que tiene en mano, acciones tesoros y compras posibles, así como las referencias a las reglas de IA que rigen al jugador tanto de compra como de juego. Además, se han implementado aquí todas las funciones que pudieran resultar de interés a la IA por la que se rige el jugador y por supuesto las acciones que tuviera que realizar un jugador para interactuar con el juego, tales como jugar acción, descartar carta, jugar tesoros, comprar cartas, etc. También podemos encontrar todas las funciones necesarias para obtener información no trivial de la partida, cobres jugados, cartas en el mazo, cartas en juego, cartas en la mano.
- **DomCard:** Define la carta modelo de Dominion. Se ha diseñado de manera que la carta guarda una referencia al poseedor así como a un enumerado DomCardName que analizaremos más adelante. Esta clase DomCard actúa como modelo para el resto de cartas que heredarán de ella, definiendo los métodos y sus implementaciones por defecto. Cada carta individual re-escribirá el método play() que será donde se implementen los efectos de la carta en el juego.
- **DomEngine:** Es la clase que organiza la simulación más allá del funcionamiento del juego. En esta clase se definen los métodos para la gestión de IAs(Bots), así como se crea el log de la simulación y los correspondientes gráficos. Además, es el encargado de comenzar la simulación con los datos que obtiene por parámetro desde la interfaz gráfica. Contiene el main del proyecto y será la clase que inicializa todo el funcionamiento, creando la GUI que verá el usuario.

- **DomBuyRule:** Esta clase encapsula las reglas de compra y sus métodos de funcionamiento. Representa un archivo XML que define una serie de cartas, prioridades y condiciones que modifican las decisiones que toma un bot en la simulación. El conjunto de esas reglas serán las que formen una estrategia concreta. Esta clase define una de esas reglas por la cual el bot intentará comprar una carta siempre y cuando se cumplan una serie de condiciones definidas por la clase DomBuyCondition.
- **DomBuyCondition:** Esta clase combinada con DomBuyRule son las que definen la estrategia de compra de un bot. A la hora de definir una regla de compra, esta estará compuesta por una serie de condiciones que se definen en esta clase.

- **DominionSimulator.Cards:** Este paquete contiene todas las clases de cartas que serán usadas en el juego, las originales de Dominion y todas sus expansiones. Cada clase concreta redefinirá los métodos heredados necesarios para actuar de la misma manera que la carta que representa, así como creará nuevos si fuese necesario.
- **DominionSimulator.Enums:** En este paquete se almacenan los enumerados definidos para el juego, comparadores, funciones para las condiciones de compra, estrategias, y quizás lo más relevante; las cartas de forma independiente y las expansiones con las correspondientes cartas que posee cada una. Esto supone que a la hora de crear una carta deberemos introducirla en su correspondiente set además de crearla individualmente.
  - **DomCardName:** Es el enumerado que complementa a la clase DomCard que contiene los métodos y los atributos más relevantes de la carta de cara a la IA. Almacena las prioridades de juego y descarte así como las estrategias concretas que pudiera necesitar cada carta. Además actúa como una mini base de datos, cargando cada carta con sus parámetros específicos.
- **DominionSimulator.GUI:** Este paquete tiene todas las clases que forman parte de la interfaz gráfica del simulador, desde los paneles a los botones y los diferentes menús.
  - DomGui: Es la clase más importante del paquete. Contiene e inicializa todos los demás elementos de la interfaz gráfica así como hace de contacto entre los inputs del usuario y el motor del juego definido en la clase DomEngine.

- **Diseño de funcionamiento:** Primero resumiremos el funcionamiento en diferentes pasos básicos y más tarde analizaremos cada uno por separado comentando también las clases que participan en dicho paso.
    - **Funcionamiento básico:**
        - La interfaz gráfica carga el motor del juego y este inicializa la simulación.
        - Los jugadores y el tablero se resetean del juego anterior.
        - Los jugadores juegan sus turnos hasta que acaba la partida.
        - Cada jugador en su turno resuelve los efectos duraderos.
        - Cada jugador en su turno juega sus acciones siempre y cuando tenga acciones restantes para jugar y siempre siguiendo el número de prioridad de juego de las cartas posibles. En el momento de jugar una carta, esta ejecuta su propia implementación de uso.
        - El jugador jugará sus tesoros y comprará cartas siguiendo las reglas de compra asignadas hasta que no tenga más compras posibles.
        - El jugador descarta todas sus cartas de la mano y jugadas al final de cada turno.
        - Durante el juego todas las estadísticas se guardan para generar los grafos y los tiempos medios.

    - **Inicio de la simulación:** Es la primera parte del funcionamiento básico e involucra tanto a la interfaz gráfica como a la clase DomEngine. Cuando se inicializa el programa compilado, se ejecuta el main de DomEngine que crea la interfaz gráfica con la que interactuaremos con el programa, seleccionaremos las opciones deseadas así como las estrategias, las predefinidas o las que nosotros editemos o creemos a través de la interfaz. Después, una vez seleccionadas las opciones, arrancaremos la partida creando un juego con dichas opciones e inicializando todo el proceso de simulación. Desde la interfaz se llama a DomEngine y este llamará a DomGame que inicializará definitivamente la simulación, incluyendo las clases DomBoard(tablero) y DomPlayer(jugadores).
    - **Los jugadores y el tablero se resetean de la partida anterior:** El objetivo de resetear siempre estos valores es el de crear un proceso de lanzamiento de partidas homogéneo, que trate indistintamente la primera partida del resto, de manera que se puedan lanzar partidas infinitas que funcionen de manera idéntica. Cuando engine activa la simulación, al entrar en el bucle de partidas se resetean jugadores y se crea un tablero, si es la primera iteración. Al acabar cada partida se resetea el tablero y se vuelve a empezar.
    - **Los jugadores juegan sus turnos hasta que acaba la partida:** En la clase DomGame se gestionan todos los turnos de juego y en cada iteración se mira si un jugador tiene pre-turnos asignados y turnos extra. Así con todos los jugadores hasta que se acabe la partida. Esto se comprueba automáticamente al final de cada turno de un jugador.
    - **Cada jugador en su turno resuelve los efectos duraderos:** Al principio del turno de jugador durante la fase de acción se resuelven los efectos de duración que hubiera

con anterioridad. Esto se consigue marcando la carta para que no sea eliminada en el mantenimiento anterior.

- **Cada jugador en su turno juega sus acciones:** En cada turno el jugador jugará tantas acciones como pueda. Las acciones que tiene en mano serán ordenadas en función de su atributo de prioridad de juego y una vez ordenadas se comprueba si la carta debe ser jugada o no. Por defecto siempre será sí, pero algunas cartas especiales re-implementan este método para no ser jugadas en momentos específicos del juego. Cuando una carta es jugada se invoca su implementación de play() que define su efecto en la partida cuando es jugada como acción, antes de ser ejecutados estos efectos la carta es descartada de la mano del jugador y se añade a la lista de cartas en juego.
- **Cada jugador en su turno juega sus tesoros y compra cartas:** En la siguiente etapa del turno, el jugador jugará las cartas de tipo tesoro de la misma manera que las acciones, obteniéndolas todas, comprobando que son tesoros y jugándolas si la carta debe ser jugada. Una vez jugados los tesoros, se comprarán tantas cartas como el jugador pueda a través del método makeBuyDecision(). En este método se obtienen todos los objetos "BuyRule" ordenados por orden de prioridad. Por lo tanto, la primera norma corresponde a la carta que en caso de que se cumpliesen sus condiciones de compra asociadas "BuyConditions" debería ser comprada.
  Para que las condiciones de esa BuyRule sean evaluadas, primero se comprueba que no sea de tipo Prize y que se tiene dinero suficiente para comprar esa carta. En caso de que no se cumplan, la iteración pasa a la siguiente regla de compra.
- **Cada jugador al final de su turno entra en fase de mantenimiento:** Al final del turno el jugador resuelve todos los efectos de la fase de mantenimiento, limpia las variables y manda a la pila de descartes todas las cartas jugadas y las de la mano que no se han jugado, además de todas las cartas que se apartaron por causa de algún efecto; además de robar las cartas para el turno siguiente.
- **Se guardan las estadísticas:** En cada movimiento, la clase DomPlayer, DomDeck o algunas cartas concretas llaman al método addToLog() de la clase DomEngine, para ir registrando las acciones en el log. Además al final de cada partida, la clase DomEngine toma medidas de todos los tiempos de ese juego en concreto y los añade al log.

- **Diseño de la IA:** El sistema de juego tiene una IA diseñada que se basa en dos aspectos principales: la lista de prioridades externa indicada por las reglas de compra (que solo tendrá relevancia a la hora de comprar cartas) y los atributos de prioridad que se establecen a la hora de crear cada carta (prioridad de juego, descarte y eliminación).
    - **Prioridad externa:** Viene determinada por el orden en el que establezcamos las reglas de compra y, será el orden en el que el juego las analizará para evaluar si es posible realizar dicha compra y si cumple las condiciones impuestas necesarias. Esta lista de prioridades es fácilmente modificable por el usuario gracias a la interfaz de gestión que ofrece el software al contrario que la lista de prioridad interna que necesita de conocimientos de programación y de un análisis más detallado, ya que sus repercusiones en el juego son diferentes.
    - **Prioridad interna:** Depende directamente de los parámetros de las cartas y se subdivide en diferentes prioridades regidas por diferentes parámetros.
        - Prioridad de juego: Este parámetro sirve para establecer el orden en el que se jugarán las cartas- Este número no solo se usará a la hora de determinar en qué orden se juegan las acciones si no también los tesoros. Las cartas de tipo Villa que otorgan más acciones tendrán más prioridad, mientras que los terminales tendrán una prioridad mucho menor ya que no permiten continuar jugando acciones. Algunas cartas tienen un índice dinámico que varía dependiendo de la situación de la partida.
        - Prioridad de descarte: Este parámetro sirve para establecer el orden en el que las cartas serán descartadas. Siendo las cartas de Victoria y de Curse las que tengan una prioridad más alta, al igual que las cartas poderosas como posesión tendrán un índice muy bajo. Algunas cartas tienen un índice dinámico que varía dependiendo de la situación de la partida.
        - Prioridad de eliminación: Un número que determina en qué orden deben ser eliminadas las cartas en caso de ser afectados por algún efecto. El número es el mismo que el de descarte salvo para cartas de victoria o de curses. También puede ser determinado dinámicamente como en Ducado o en Estado.
        - Las cartas pueden tener más de un tipo, con esto conseguimos poder darle más posibilidades a las BuyRules.
        - ForbidenCards: Algunas cartas como contrabandistas pueden imponer que algunas cartas no se puedan comprar. Además, esta lista puede ser usada en alguna carta como efecto complejo para que después de una acción no se adquiera una carta concreta.
        - WantsToBePlayed: Todas las cartas por defecto querrán ser jugadas, pero aun así, en el momento de seleccionar una carta para jugarla en la fase de acción siempre se consulta, ya que se puede implementar en las cartas este método para que solo sean jugadas cuando se cumplan o no se cumplan ciertas condiciones del juego, independientemente de su prioridad de juego.

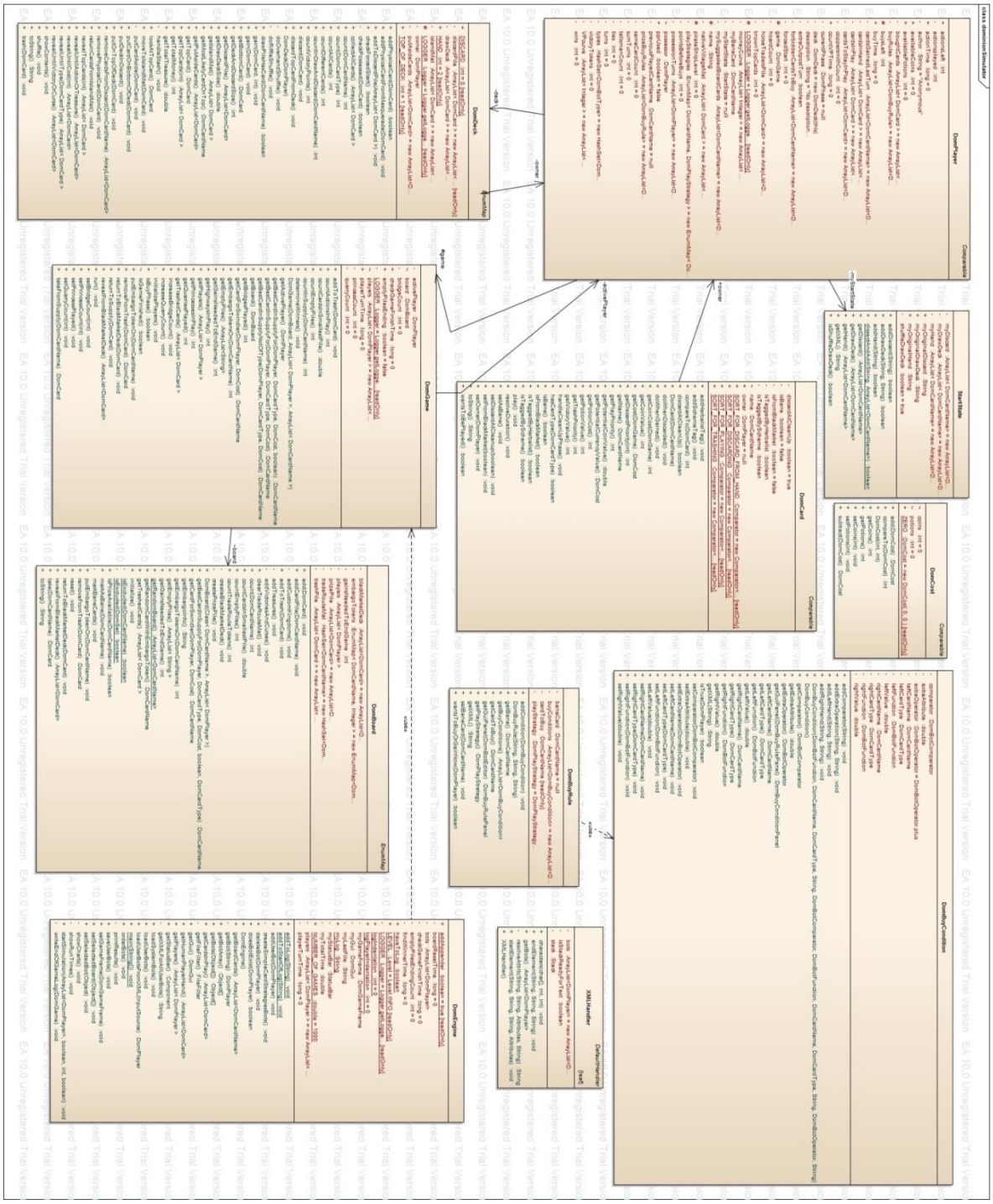


Ilustración 14. Paquete Dominion Simulator

- Implementación:
  En este apartado analizaremos la implementación específica de cada clase que compone el simulador agrupando las mismas por el paquete al que pertenecen.
    - **dominionSimulator:** Es el paquete raíz del proyecto y además engloba las clases básicas del juego.
        - DomGame: Clase encargada de la coordinación de partidas.
            - Constructor: Implementado manualmente y con 2 parámetros útiles de entrada, que son el tablero, la lista de jugadores y la lista de CardNames. Se encargará de llamar al método que inicializa los jugadores.
            - Atributos: Los más importantes son la referencia al tablero y la lista de tipo ArrayList de jugadores que toman parte en la partida.
            - Métodos públicos: Son los métodos que invocan las cartas y los jugadores para acceder al tablero.
            - Métodos privados: Solo tiene 1, que es invocado por el constructor de esta clase para inicializar los jugadores.
        - DomBoard: define el tablero del juego y por lo tanto gestionará todas las acciones relacionadas con el manejo de cartas. Hereda de EnumMap y por lo tanto actúa como una clase contenedora.
            - Constructor: Un constructor implementado que llama al súper constructor de la clase EnumMap y que se le pasa por parámetro la lista de jugadores así como los enumerados que serán las claves del EnumMap.
            - Atributos: Las pilas de cartas, que son parte de la clase en sí al heredar de EnumMap, así como las listas de eliminadas, blackMartket y Prizes, que son del tipo ArrayList. Un contador que refleja el número mínimo de cartas a obtener con las que se podría acabar la partida.
            - Métodos públicos: Gestión de las pilas y de las cartas, invocados por la clase DomGame.
            - Métodos privados: Para corroborar si un set de cartas o una carta están excluidas.

        - DomPlayer: Esta clase representará al jugador en el simulador.
            - Constructor: Son dos constructores manuales con atributos String que indicará el nombre del jugador. El otro constructor incluye también el nombre del autor.
            - Atributos: Esta clase tiene una serie de listas de cartas referentes a las reglas de compra que seguirá la IA. Además, cuenta con listas que contienen las cartas del jugador, tanto las que tiene en mano como las que están en juego o las que se deben quedar en el juego. Todas estas listas son del tipo ArrayList. Además de contar con todos los de tipo *int* en referencia a las estadísticas del jugador en la partida que juega y en referencia a simulación total.

- Métodos públicos: Prácticamente todos los que obtienen la información necesaria para que la IA tome decisiones así como las funciones que puede realizar un jugador en la partida.
- Métodos privados: Los que tienen que ver con la activación de las diferentes fases de un turno (acción, compra, mantenimiento) así como la resolución de efectos o el reinicio de variables.

- DomCard: Define la carta modelo de Dominion.
  - Constructor: Un constructor que recibe por parámetro un enumerado de tipo DomCardName que hará referencia al modelo de carta.
  - Atributos: Son mayormente *boolean* para hacer comprobaciones en diferentes partes de la partida.
  - Métodos públicos: Prácticamente todos los que obtienen la información necesaria para que la IA tome decisiones así como las funciones que puede realizar un jugador en la partida.
  - Métodos privados: Los que tienen que ver con la activación de las diferentes fases de un turno (acción, compra, mantenimiento) así como la resolución de efectos o el reinicio de variables.
- DomEngine: Es la clase que organiza la simulación más allá del funcionamiento del juego, que además contiene el método main.
  - Constructor: Constructor que carga los bots y además crea y establece la referencia a la interfaz gráfica en el atributo.
  - Atributos: Tiene dos listas ArrayList con los jugadores así como con los bots de la simulación además de los atributos de tipo *long* que guardaran los tiempos de la simulación.
  - Métodos públicos: El main que es el que arrancara todo el sistema, así como todos los métodos de gestión de bots.
  - Métodos privados: Crear un bot estándar u obtener un bot concreto de la lista por nombre.
- DomBuyRule: Esta clase encapsula las reglas de compra y sus métodos de funcionamiento.
  - Constructor: El constructor tiene tres parámetros String. Uno corresponde al nombre de la carta, otro al de la estrategia de juego y el último identifica la carta de Bane[7].
  - Atributos: Tiene cuatro atributos. El primero hace referencia a la carta que se pretende comprar siempre y cuando se cumplan las condiciones de compra de tipo DomBuyConditions almacenadas en otro atributo de tipo ArrayList. Además, almacenamos la estrategia de juego y la carta de Bane.

[7] La Carta Bane se juega para evitar el efecto de la bruja joven

- Métodos públicos: Para obtener información sobre la regla de compra, para añadir condiciones de compra y para establecer la carta de tipo Bane.
- Métodos privados: No tiene.

▪ DomBuyCondition: Establece las condiciones para que una BuyRule se cumpla.

- Constructor: Posee el constructor por defecto y además otro con una serie de atributos de tipo DomCardName, DomCardType y DomBotComparator a través de los cuales se puede establecer una expresión booleana de comparación de cartas y sus atributos.
- Atributos: Todos los referentes a las condiciones de compra, desde evaluadores a cartas, sus tipos e incluso datos numéricos necesarios para la regla.
- Métodos públicos: Todas los referentes al manejo de la condición, desde la obtención de información hasta establecer nuevos atributos diferentes a los establecidos al crear la condición.
- Métodos privados: No tiene.

o **DominionSimulator.Cards:** En este paquete están definidas todas las cartas del juego de manera unitaria.

▪ Carta genérica: Cualquiera de las cartas del juego que hereda de DomCard.

- Constructor: Llama al constructor de su superclase introduciendo por parámetro su enumerado DomCardName específico.
- Atributos: Los de su superclase.
- Métodos públicos: En su mayoría reescriben algunos de su superclase, sobre todo los referentes a prioridades y sus efectos en el juego.
- Métodos privados: Algunas cartas incluyen métodos privados de uso interno.

o DominionSimulator.Enums: En este paquete se almacenan los enumerados definidos para el juego, explicados previamente en la parte de diseño. En su mayoría, los enumerados de este paquete no incorporan métodos específicos salvo el toString(), que sirve para escribir en el log de la partida.

▪ DomCardName: Es un enumerado muy amplio que define todas y cada una de las cartas del juego.

- Constructor: Establece los costes y prioridades de la carta en concreto.
- Atributos: Almacena los costes y las prioridades de la carta.
- Métodos públicos: Obtención de costes, prioridades y otros métodos propios del manejo de cartas.
- Métodos privados: No tiene.

- DominionSimulator.GUI: Este paquete tiene todas las clases que forman parte de la interfaz gráfica del simulador.
  - DomGui: Es la clase principal de la interfaz y contiene el resto de clases del paquete.
    - Constructor: Establece la referencia a la clase DomEngine, dicha referencia se pasa por parámetro al constructor. Además, de inicializar todos los componentes de la interfaz.
    - Atributos: La lista de botones, agrupados por el tipo de funcionalidad. Las diferentes etiquetas así como los diferentes paneles que tiene la interfaz.
    - Métodos públicos: Los referentes a establecer información de los gráficos de la simulación así como a la muestra de dichos datos.
    - Métodos privados: Todos los referentes al manejo de la interfaz, botones, paneles, barras, etc.…

1.5 Análisis final

Tras analizar los productos o proyectos más importantes en el mercado similares al nuestro o relacionados con Dominion, procederemos a hacer un análisis de los puntos débiles y fuertes de los dos simuladores que "competirían con el nuestro". A la vez, que sacaremos unos puntos o conclusiones generales extraídas del análisis del estado del arte al completo.

1.5.1 Puntos fuertes

Este apartado se centrará en analizar los puntos fuertes o positivos de los dos simuladores analizados. El interés u objetivo de este análisis es que dichos puntos fuertes puedan repercutir en el proyecto a desarrollar, aprendiendo de lo que se ha sido realizado ya con anterioridad.

1.5.1.1 Dominiate

- Ligereza del código.
- Implementación web.
- Formato de creación de cartas, modelos básicos bien diferenciados y herencia posterior.
- Posible implementación de estrategias mediante la interfaz web de manera directa.
- Constante actualización del estado de la partida siempre con anterioridad a que la IA tome una decisión.

1.5.1.2 Geronimoo

- Ni jugadores ni cartas acceden al tablero directamente, siempre a través de la clase DomGame.
- Estrategias almacenadas en XML, facilitando la modificación manual y no necesariamente a través de interfaces.
- Sistema de prioridades diferenciado para cada tipo acción (Jugar, descartar, eliminar).
- Prioridades complejas en las compras(prioridad + condiciones)
- Gestión de suministros integrada en la propia clase, gracias a qué tablero hereda de EnumMap.
- Búsqueda de cartas por nombre gracias a la búsqueda usando enumerados como claves.
- Interfaz para creación de reglas de compra.
- Simulación eficiente.
- Sistema de análisis de las simulaciones efectivo.
- Diseño del lanzamiento de partidas bien estructurado, para que cada módulo o clase funcione de manera independiente pero a la vez bien comunicada con el resto del sistema.

1.5.2 Puntos débiles

1.5.2.1 Dominiate

- Lenguaje poco común, Coffeescript es una adaptación de JavaScript.
- Complicaciones para arrancarlo en formato consola.
- La interfaz Web solo admite 2 jugadores simultáneamente.
- Dificultad en la creación de nuevas cartas.
- No poder heredar de varios modelos.
- Imposibilidad de creación de estrategias para usuarios normales o sin conocimientos de programación.
- Rigidez de la IA.

1.5.2.2 Geronimoo

- Dificultad para adaptar simulador a juego, introduciendo la posibilidad de que un humano actúe como uno de los jugadores.
- Tener todas las estrategias en un mismo XML dificulta su acceso y modificación además del análisis manual.
- Sistema de prioridades rígido para tipos de acción concreto (jugar, descartar, eliminar).
- El sistema de prioridades dependiente de cada carta no es la forma más efectiva de evaluación ya que no te permite evaluar una carta por lo que realmente hace sino la carta en sí misma y tiene muy poca flexibilidad.
- Al ser los suministros parte de una clase y actuar como objetos de un mapa pierden independencia ya que no pueden ser tratados como objetos independientes al no tener una clase propia.
- Se crean objetos por cada carta, ocupando mucho más espacio del que sería necesario.
- Carece de interfaz de modificación de cartas.
- Simulación eficiente pero poco interactiva.

- Es bastante costoso añadir nuevas cartas, no solo su implementación si no a la vez darles la prioridad adecuada para que funcione en consonancia con el funcionamiento del sistema de evaluación en el resto de cartas.

1.5.3 Conclusiones generales

Una de las primeras conclusiones a las que hemos llegado es que debíamos separar la interfaz del motor del juego y de las inteligencias o usuarios; Siguiendo el modelo de los simuladores de ajedrez y dándole un punto más de independencia a cada uno de los módulos.

Siempre que se usen bases de datos deben estar externalizadas a través de SQL o archivos XML evitando lo máximo posible la dependencia del código.

Si bien en la rigidez en el sistema puede ser beneficioso de cara al programador en cuestión rapidez, a la larga puede ser contraproducente ya que pudiera darse el caso de modificaciones de funcionamiento inquiriendo un posible rediseño y reprogramación del producto completo.

A la hora de diseñar una IA hay que intentar que haga una comparación o evaluación de las posibilidades lo más real posible, es decir, que la evaluación sea lo suficientemente profunda y escalable posible para conseguir que el abanico de posibilidades a evaluar pueda aumentar sin por ello perder capacidad de raciocinio.

Es interesante ofrecer una interfaz de diseño de IAs amigable para el usuario medio, como pasa en el simulador Geronimoo o en algunos simuladores de Ajedrez.

2 Objetivos

En este apartado de la memoria resumiremos cuales han sido las motivaciones para realizar este proyecto, tanto personales como profesionales. A la vez marcaremos los objetivos específicos que tanto el tutor como yo nos planteamos conseguir a través de este simulador.

A modo introductorio, señalar que el objetivo básico del proyecto era conseguir crear una plataforma para Dominion que permitiera jugar a humanos e IAs indistintamente. A la vez, diseñarlo e implementarlo de la manera lo más flexible posible permitiendo ampliaciones y modificaciones posteriores llegando incluso a adaptarlo para correr modificaciones de Dominion u juegos nuevos bajo la misma plataforma.

2.1 Motivaciones del proyecto

Los hogares están cada vez más informatizados, sistemas inteligentes de calefacción, cocinas y electrodomésticos inteligentes, alarmas con sensores cada vez más complejos…

Todo esto tiene un punto en común y es la inteligencia artificial presente a través de muchas formas, sistemas de reconocimiento, sistemas de lógica difusa, agentes autónomos. Inteligencia artificial que no solo es usada por el usuario medio sino que también tiene cabida en el mundo profesional, sea analizando inversiones de bolsa, [8]ayudando a los médicos en sus diagnósticos[9] o en sistemas de detección de problemas electrónicos[10].

Cuando se planteó la posibilidad de realizar una plataforma que pudiera servir para la investigación a la vez que para el entretenimiento del usuario medio no hubo dudas al respecto, este era el proyecto. En la situación del estado del arte en la cual nos encontramos simuladores puros sin cabida para la interacción humana y con muy pocas posibilidades de ampliación, encontramos un hueco para crear una plataforma nueva que no solo le diera más libertad al desarrollador, sino que también permitiera al usuario medio participar del mismo.

Para conseguir esto tendríamos que diseñar un modelo que permitiera jugar partidas completas de Dominion tanto a IAs como a personas de manera simultánea y que además fuera de fácil extensión. Además, creímos necesario separarnos del resto aplicaciones (simuladores) vistas en el estado del arte. Partiendo de que dichos simuladores contienen un modelo de IA que es totalmente específico (sistemas basados en reglas) está fuertemente incrustado en su código.

[8] http://www.eleconomista.es/mercados-cotizaciones/noticias/2423963/09/10/-Indra-desarrolla-una-aplicacion-para-predecir-la-evolucion-de-los-valores-en-bolsa.html
[9] http://www.agenciasinc.es/Noticias/Inteligencia-artificial-que-ayuda-en-el-diagnostico-de-enfermedades-graves
[10] http://www.hindawi.com/journals/vlsi/2008/630951/

Nuestra motivación es resolver los problemas u objetivos marcados a través de una plataforma que modele el juego de forma genérica, separando la implementación de las reglas del juego de la lógica de juego o toma de decisiones. Consiguiendo desde el punto de vista del usuario; una plataforma totalmente funcional para Dominion. En la que las IAs que se implementaran pudieran jugar partidas completas reales, tanto con otras IAs como con o entre usuarios humanos. Y consiguiendo desde el punto de vista del investigador/desarrollador un sistema en el que la implementación de la IA no tuviera que atender necesariamente a un modelo predefinido. Sino por el contrario que cualquier desarrollador tuviera prácticamente plena libertad de diseño sin por ello tener que modificar el resto del sistema.

Además, a título personal, pese al interés por la investigación, siempre me he considerado más un diseñador que un investigador con lo cual considero que un proyecto que tuviera una parte centrada en la innovación e investigación pero que estuviera fuertemente basado en el diseño, como si un producto software se tratase sería lo más adecuado para mis gustos y capacidades.

2.2 Objetivos

Con este proyecto nos planteamos realizar un simulador/plataforma interactiva con las siguientes características:

- **Una plataforma versátil:** El objetivo es conseguir una plataforma con una estructura fácilmente modificable o adaptable a futuros cambios; Tanto para modificaciones importantes (incluso otros juegos similares a Dominion) como para extensiones del mismo (debidas a ampliaciones oficiales).
- **Una base de datos accesible y bien estructurada**: Esto se consigue usando un formato estándar (XML o MySQL) y accesible al usuario medio sin conocimientos de programación. En la medida de lo posible usar diferentes archivos o tablas para crear una estructura más clara.
- **Facilitar el razonamiento de las IAs:** Se persigue crear un diseño que permita que las IAs sean lo más inteligentes/coherentes a la hora de evaluar las diferentes acciones a realizar; dándole la posibilidad de decidir sobre el máximo de opciones posibles y con la máxima información posible.
- **Una interfaz independiente del motor del juego:** Que posibilite un funcionamiento independiente de la entrada y salida de información y órdenes del motor. Posibilitando que la gestión de dicha información por parte de la interfaz sea opaca o indiferente para el motor del juego y viceversa de manera que el motor trate indistintamente un jugador controlado por un ser humano o por una inteligencia artificial. Siguiendo el patrón Modelo Vista Controlador.
- **Una implementación de Dominion totalmente funcional**: El objetivo es que la implementación sea funcional a la vez que completa. Con todo el set de cartas pertenecientes al juego básico, con sus correspondientes efectos implementados además de todas sus reglas de funcionamiento que hagan un simulador fidedigno.

3 Desarrollo

En este capítulo de la memoria definiremos el proceso que hemos llevado a cabo para diseñar los módulos o subsistemas que componen nuestro proyecto. Empezaremos con la fase de análisis que dio lugar al posterior diseño del sistema. A través de los requisitos que deben cumplir el sistema y sus derivados casos de uso intentaremos dar una visión muy clara de las bases en las que se apoya el diseño final del programa.

3.1 Análisis de requisitos

3.1.1 Requisitos funcionales

Estos requerimientos definirían a gran escala nuestro futuro sistema, las funcionalidades que debe ofrecer así como los sus funciones específicas de entrada y salida, manejo de excepciones, etc.

Identificador	RF1
Descripción	El usuario debe poder crear y eliminar los jugadores desde el menú principal.
Necesidad	Esencial.

Tabla 1. RF1 – Gestión de jugadores de la partida

Identificador	RF2
Descripción	El usuario debe poder seleccionar si un jugador es humano o estará controlado por una IA.
Necesidad	Esencial.

Tabla 2. RF2 – Selección de controlador de jugador

Identificador	RF3
Descripción	El usuario debe poder elegir cuál será la plantilla de juego que define que cartas son las que serán accesibles en la partida.
Necesidad	Deseable.

Tabla 3. RF3 – Elección de plantilla de juego

Identificador	RF4
Descripción	El usuario debe poder lanzar la partida si el número de jugadores es correcto.
Necesidad	Esencial.

Tabla 4. RF4 – Gestión de jugadores de la partida

Identificador	RF5
Descripción	El usuario debe poder salir del juego desde el menú principal.
Necesidad	Esencial.

Tabla 5. RF5 – Selección de controlador de jugador

Identificador	RF6
Descripción	El usuario debe poder elegir cuál será la plantilla de juego.
Necesidad	Deseable.

Tabla 6. RF6 – Elección de plantilla de juego

Identificador	RF7
Descripción	El sistema debe poder mostrar por consola todas las plantillas de cartas disponibles.
Necesidad	Deseable.

Tabla 7. RF7 – Elección de plantilla de juego

Identificador	RF8
Descripción	El usuario debe poder crear y editar plantillas desde la interfaz.
Necesidad	Esencial.

Tabla 8. RF8 – Gestión de jugadores de la partida

Identificador	RF9
Descripción	El sistema debe poder listar todos los archivos de la base de datos del juego tanto el repositorio de cartas como los ficheros de plantilla de juego.
Necesidad	Esencial.

Tabla 9. RF9 – Listado de ficheros de la BBDD

Identificador	RF10
Descripción	El sistema debe diferenciar, agrupar y listar las cartas de la plantilla por sus tipos.
Necesidad	Deseable.

Tabla 10. RF10 – Elección de plantilla de juego

Identificador	RF11
Descripción	El usuario debe poder salir del menú de edición de plantillas sin salir del juego.
Necesidad	Esencial.

Tabla 11. RF11 –Selección de controlador de jugador

Identificador	RF12
Descripción	El usuario debe poder introducir todos los comandos por teclado.
Necesidad	Esencial.

Tabla 12. RF12 – Selección de controlador de jugador

Identificador	RF13
Descripción	El sistema debe poder mostrar todas las opciones por consola.
Necesidad	Esencial.

Tabla 13. RF13 – Control de número de jugadores

Identificador	RF14
Descripción	El sistema debe representar el juego de Dominion de manera fidedigna. Es decir, que esta adaptación digital sigue todas las reglas y mecanismos del juego original.
Necesidad	Esencial.

Tabla 14. RF14 – Elección de plantilla de juego

Identificador	RF15
Descripción	El sistema debe saber siempre qué jugador tiene el turno actual.
Necesidad	Esencial.

Tabla 15. RF15 – Elección de plantilla de juego

Identificador	RF16
Descripción	El sistema debe imprimir por pantalla el orden clasificatorio al acabar la partida.
Necesidad	Deseable.

Tabla 16. RF16 – Elección de plantilla de juego

3.1.2 Requisitos no funcionales

Los requisitos no funcionales definen los atributos o propiedades del sistema así como algunas restricciones de diseño. También se les conoce como requisitos de calidad ya que algunos de ellos establecen unos mínimos que garantizan de eficiencia, efectividad, fiabilidad, robustez de las propias funciones definidas en los requisitos funcionales.

Identificador	RNF 1
Descripción	El ordenador debe tener instalado Java 6 o superior.
Necesidad	Esencial.

Tabla 17. RNF1 – Versión de Java

Identificador	RNF 2
Descripción	El ordenador que ejecute el sistema debe tener un SO que soporte una máquina virtual de Java.
Necesidad	Esencial.

Tabla 18. RNF2 –SO válido

Identificador	RNF 3
Descripción	El sistema debe estar divido en paquetes que agrupen clases similares.
Necesidad	Deseable.

Tabla 19. RNF3 – División en paquetes

Identificador	RNF 4
Descripción	La interfaz debe ser totalmente independiente del resto del sistema.
Necesidad	Esencial.

Tabla 20. RNF4 – División interfaz – plataforma

Identificador	RNF 5
Descripción	El sistema debe asegurarse de que el mínimo y el máximo de jugadores se cumplen.
Necesidad	Esencial.

Tabla 21. RNF5 – Control de número de jugadores

	RNF6
Descripción	La base de datos de cartas debe estar enteramente contenida en archivos externos al código.
Necesidad	Deseable.

Tabla 22. RNF6 – Base de datos externa

Identificador	RNF7
Descripción	Debe existir un fichero específico e independiente para las cartas accesibles en el juego
Necesidad	Esencial

Tabla 23. RNF7 – Control de número de jugadores

Identificador	RNF8
Descripción	Los ficheros de plantillas deben ser independientes entre sí a la vez que suficientes para representar un modelo de partida.
Necesidad	Esencial.

Tabla 24. RNF8 – Elección de plantilla de juego

Identificador	RNF9
Descripción	Todos los atributos de la carta deben estar definidos en el archivo de la base de datos.
Necesidad	Esencial.

Tabla 25. RNF9 – Elección de plantilla de juego

Identificador	RNF10
Descripción	Todos los atributos relativos a cada suministro deben estar definidos en el archivo de plantilla de la base de datos.
Necesidad	Esencial.

Tabla 26. RNF10 – Elección de plantilla de juego

Identificador	RNF11
Descripción	Todas las cartas del juego deberán estar definidas por listas de efectos independientes.
Necesidad	Deseable.

Tabla 27. RNF11 –Elección de plantilla de juego

Identificador	RNF12
Descripción	El motor debe gestionar las cartas de manera que en el sistema solo exista una única representación de la carta, evitando que existan objetos que representen la misma carta.
Necesidad	Esencial.

Tabla 28. RNF12 – Elección de plantilla de juego

Identificador	RNF13
Descripción	El sistema debe aplicar los efectos de manera independiente siendo estos autónomos con respecto al resto de efectos y externos con respecto a la carta que los porta.
Necesidad	Esencial.

Tabla 29. RNF13 – Elección de plantilla de juego

3.2 Casos de uso

A continuación se muestran los Casos de Uso derivados de los Requisitos de Usuario que se han definido en el anterior apartado.

A modo de recordatorio mencionar que los casos de uso son una técnica de escenarios incorporada en UML que describe la interacción entre los actores y el sistema. Un conjunto de casos de uso describe todas las posibles interacciones con el sistema. Describen lo que puede ir mal y cómo manejar el problema. Los hemos dividido en 3 categorías diferentes de manera que cada una de ellas agrupe funcionalidades o procesos similares, para poder facilitar su entendimiento y desarrollo.

3.2.1 Gestión de Partidas

En el siguiente diagrama que podemos ver en la ilustración 15, analizaremos las diferentes situaciones o procesos que se encontrará un usuario a la hora de gestionar/lanzar una partida en el simulador. Algunos casos de uso son derivados directamente de alguno más grande y en cierta manera completan el caso de uso.

Gestionar jugadores comprende todas las funciones relacionadas con la lista de jugadores que tomarán parte en la partida; tanto humanos como jugadores controlados por IAs. Al lanzar partida se verifica siempre el número de jugadores mínimo y máximo que podemos definir como un caso de uso dependiente de este. Al seleccionar plantilla incluimos también el caso de uso que define el proceso de listar las plantillas disponibles accediendo a una ruta concreta del disco duro.

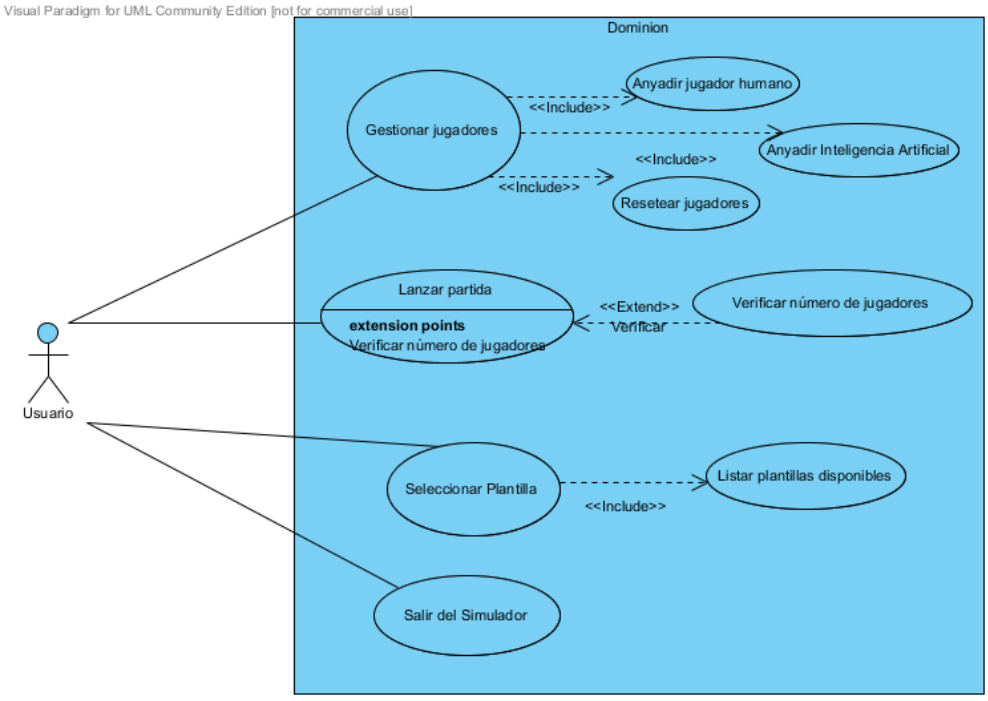


Ilustración 15. Gestión de partidas

Identificador	CU 01	
Nombre	Gestionar Jugadores.	
Actores	Usuario.	
Objetivo	Crear y agregar los jugadores a la partida a lanzar.	
Curso normal		**Alternativas**
1. El usuario selecciona la opción añadir jugador. 2. introduce el nombre del mismo por consola. 3. Y decide entre un control humano o por una IA.		a) Si en 3 introduce una opción no valida se repite.
Pre-condición	• El Sistema debe estar en funcionamiento.	
Post-condición	• Se añade un jugador más a la lista de jugadores que estarán en la partida.	

Tabla 30. CU1 –Gestionar jugadores

Identificador	CU 02
Nombre	Resetear jugadores.
Actores	Usuario.
Objetivo	Borrar la lista de jugadores de la partida.
Curso normal	1. El usuario elige la opción por consola de resetear la lista de jugadores.
Pre-condición	• El Sistema debe estar en funcionamiento.
Post-condición	• La lista de jugadores que estarán en la partida se vacía.

Tabla 31. CU2 –Resetear jugadores

Identificador	CU 03	
Nombre	Seleccionar Plantilla.	
Actores	Usuario.	
Objetivo	Seleccionar una plantilla para el juego.	
Curso normal		**Alternativas**
1. El usuario elige la opción por consola de seleccionar una plantilla para el juego. 2. El sistema lista por pantalla las plantillas disponibles. 3. El jugador introduce el id de la plantilla que quiere.		a) Si en 3 introduce una opción no valida se repite.
Pre-condición	• El Sistema debe estar en funcionamiento. • Hay una lista de plantillas disponibles.	
Post-condición	• Se cambia la plantilla por defecto por la seleccionada.	

Tabla 32. CU3 –Seleccionar plantilla

Identificador	CU 04
Nombre	Listar plantillas disponibles.
Actores	Usuario.
Objetivo	Mostrar por pantalla las plantillas disponibles.
Curso normal	**Alternativas**
1. El sistema muestra por pantalla las plantillas disponibles con un número asignado cada una.	b) Si no es un fichero plantilla no se mostrará, se comprueba la validez de los ficheros antes de mostrarlos.
Pre-condición	• El Sistema debe estar en funcionamiento. • Las plantillas deben ser archivos.XML y estar en el directorio plantillas.
Post-condición	• Las plantillas se muestran por consola.

Tabla 33. CU4 –Listar Plantillas disponibles

Identificador	CU 05
Nombre	Lanzar Partida.
Actores	Usuario.
Objetivo	Lanzar una partida con la plantilla seleccionada y los jugadores creados.
Curso normal	**Alternativas**
1. El usuario selecciona la opción de lanzar partida. 2. Se lanza una partida normal.	a) Si no se cumple el número de jugadores adecuado no se lanza partida. b) Si no se ha seleccionado una plantilla antes se usa la de por defecto.
Pre-condición	• El Sistema debe estar en funcionamiento. • La plantilla debe estar disponible. • El mínimo y máximo de jugadores debe cumplirse.
Post-condición	• Se inicia la partida.

Tabla 34. CU5 –Lanzar Partida

Identificador	CU 06
Nombre	Salir del simulador.
Actores	Usuario.
Objetivo	Salir del sistema.
Curso norma,	1. El usuario elige la opción de cerrar el sistema del menú principal.
Pre-condición	• El Sistema debe estar en funcionamiento.
Post-condición	• Se cierra el sistema.

Tabla 35. CU6 –Salir del simulador

3.2.2 Gestión de Plantillas

En este grupo, que se muestra en la ilustración 16, volvemos a encontrar casos de uso que dependen enteramente de otros o, que en cierta manera los implementan. Puede-ser crear plantilla o editar plantilla, que son las dos opciones de Gestionar Plantillas. Volvemos a ver el caso "listar plantillas disponibles" que vimos en el grupo anterior, además de un listar Cartas que si bien es un proceso por sí mismo, depende de Gestionar Pilas.

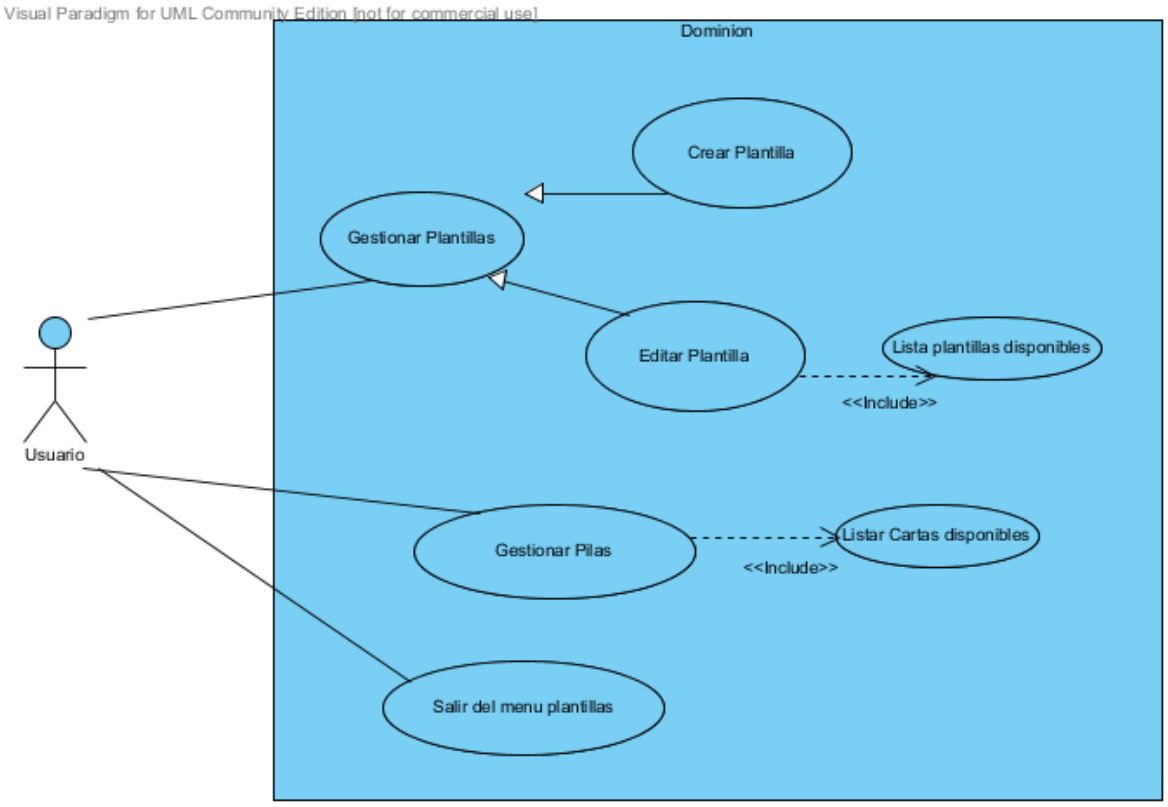


Ilustración 16. Gestionar Plantillas

Identificador	CU 07 Gestionar Plantillas	
Nombre	Gestionar Plantillas.	
Actores	Usuario.	
Objetivo	Crear y editar las plantillas de cartas accesibles en la partida del juego.	
Curso normal		**Alternativas**
1. El usuario selecciona la opción de menú de gestión de plantillas**.** 2. El usuario decide si selecciona una. plantilla existente o crea una nueva 3. El sistema muestra las opciones de gestión de la plantilla. 4. El usuario introduce la opción por teclado.		a) Si no se introduce un número correcto se repiten (se aplica a 2 y a 4). b) 2.1 El sistema lista las plantillas. disponibles en caso de edición. c) 2.2 El sistema crea una plantilla nueva.
Pre-condición	• El Sistema debe estar en funcionamiento.	
Post-condición	• Se abre el menú de edición para una plantilla existente o una de nueva creación.	

Tabla 36. CU7 –Gestionar Plantillas

Identificador	CU 08 Crear Plantilla
Nombre	Crear Plantilla.
Actores	Usuario.
Objetivo	Crear plantilla de cartas.
Curso normal	1. El usuario elige la opción por consola de crear una plantilla totalmente nueva. 2. El usuario introduce el nombre de la plantilla por consola.
Pre-condición	• El Sistema debe estar en funcionamiento. • Se debe tener acceso al disco duro.
Post-condición	• Se creará en la carpeta plantillas un archivo .XML. formateado y apto para su edición y uso. • Se abrirá el menú de edición.

Tabla 37. CU8 –Crear plantilla

Identificador	CU 09 Editar Plantilla
Nombre	Editar Plantilla.
Actores	Usuario.
Objetivo	Editar una plantilla de cartas.
Curso normal	1. El sistema lista las plantillas disponibles. 2. El usuario elige la plantilla que quiere editar.
Pre-condición	• El Sistema debe estar en funcionamiento. • Se debe tener acceso al disco duro. • El usuario ha seleccionado editar una plantilla.
Post-condición	• Se abrirá el menú de edición.

Tabla 38. CU9 –Editar plantilla

Identificador	CU 10 Gestionar Pilas	
Nombre	Gestionar Pilas.	
Actores	Usuario.	
Objetivo	Gestionar las diferentes pilas de suministro de las plantillas.	
Curso normal		**Alternativas**
1. El sistema muestra las opciones de tipos de cartas a gestionar. 2. El usuario elige el tipo de cartas que quiere gestionar de la plantilla elegida. 3. Una vez elegido el tipo se pueden ver las cartas del repositorio, pilas de ese tipo en la plantilla o seleccionar el menú decisión. 4. En el menú de edición se podrá optar por crear, editar o eliminar una pila.		a) 3.1 El sistema lista las cartas disponibles por consola. b) 3.2 El sistema muestra por consola las pilas de ese tipo concreto que ya están en la plantilla.
Pre-condición	• El Sistema debe estar en funcionamiento. • La plantilla está cargada en el sistema.	
Post-condición	• Se realizan los cambios elegidos en la plantilla.	

Tabla 39. CU10 –Gestionar pilas

Identificador	CU 11 Listar Cartas disponibles
Nombre	Listar plantillas disponibles.
Actores	Usuario.
Objetivo	Mostrar por pantalla las cartas disponibles.
Curso normal	1. El sistema muestra por pantalla las cartas disponibles del tipo seleccionado previamente que se pueden añadir a la plantilla.
Pre-condición	• El Sistema debe estar en funcionamiento. • El fichero de cartas debe estar cargado en el sistema.
Post-condición	• Las cartas se muestran por consola.

Tabla 40. CU11 –Listar Cartas Disponibles

Identificador	CU 12 Salir del menú plantillas
Nombre	Salir del menú plantillas.
Actores	Usuario.
Objetivo	Salir del menú plantillas.
Curso normal	1. El usuario elige la opción de salir del menú de gestión de plantillas.
Pre-condición	• El Sistema debe estar en funcionamiento.
Post-condición	• Se vuelve al menú inicial del simulador.

Tabla 41. CU12 –Salir del menú plantillas

3.3 Jugar Dominion

En este grupo de casos de usos hemos definido los relacionados con las interacciones del jugador con el sistema durante el turno de juego, como podemos ver en la ilustración 17. Básicamente es una toma de decisiones, pero que tiene dos comprobaciones dependiendo del caso de uso. En el caso de jugar cartas de acción, o comprar cartas se hace una comprobación ya que por defecto solo se juega o se compra una carta por turno.

Además en el momento de jugar cartas de acción o tesoro se debe comprobar que dichas cartas pertenecen a ese tipo concreto, ya que según las reglas del juego cada tipo de carta solo se puede jugar en el momento específico del turno destinado a ello.

Jugar acción es diferente de jugar tesoro pese a que el proceso de lectura de efectos, porque las acciones al igual que las compras son hechos limitados, mientras que jugar tesoros se pueda realizar de manera ilimitada. Es por eso que se han definido casos de uso diferentes

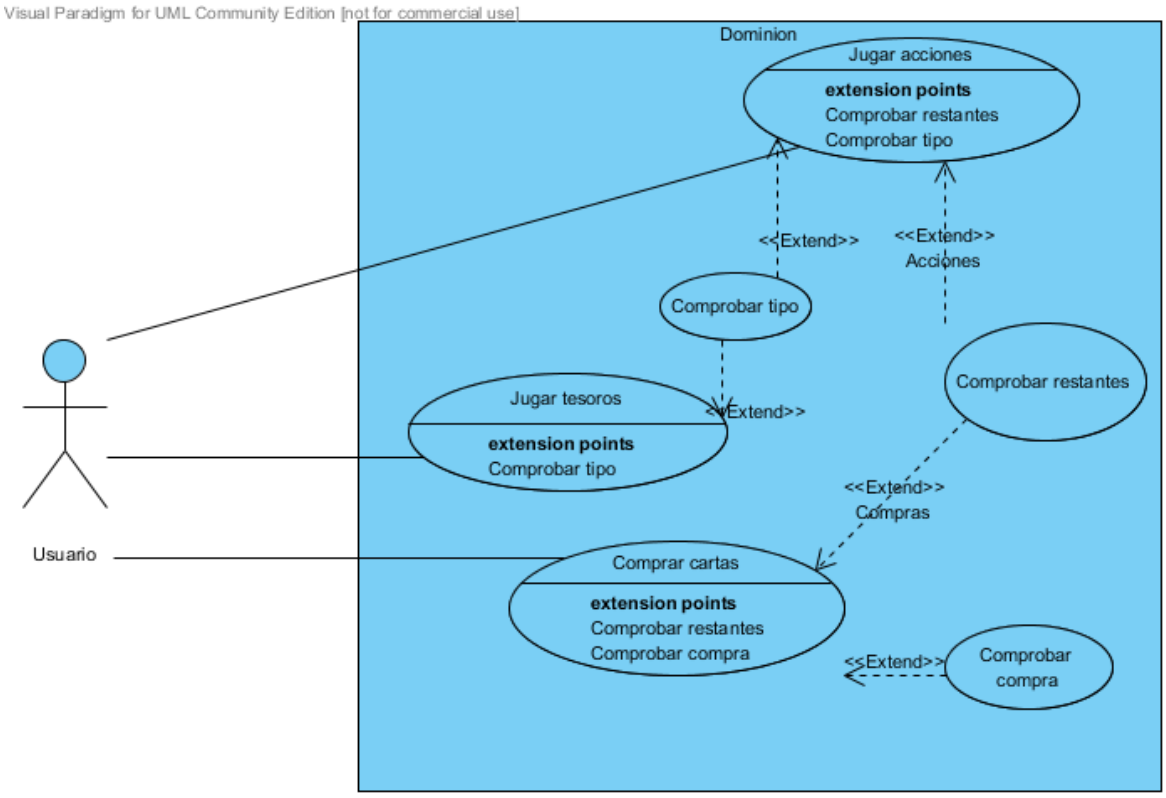


Ilustración 17. Jugar Dominion

Identificador	CU 13	Jugar Acciones
Nombre	Jugar acciones	
Actores	Usuario	
Objetivo	Jugar una carta de acción	
Curso normal		**Alternativas**
1. El sistema muestra por consola las cartas posibles a jugar 2. El jugador introduce por consola el identificador de la carta a jugar		a) 1.1 El sistema se salta la ronda porque el jugador no dispone de acciones b) 2.1 El jugador introduce la opción de no jugar acciones
Pre-condición	• El Sistema debe estar en funcionamiento • El jugador debe tener acciones restantes • El jugador está en su turno en la fase de acción	
Post-condición	• Se resta una acción a las disponibles • La carta de acción de gestiona en el tablero para la ejecución de su lista de efectos	

Tabla 42. CU13 –Jugar Acciones

Identificador	CU 14	Comprobar Restantes
Nombre	Comprobar restantes	
Actores	Usuario	
Objetivo	Comprobar cuantas acciones/compras restantes tiene el jugador	
Curso normal	1. Este proceso comprueba que el número de acciones/compras restantes es superior a 0 2. En caso afirmativo el jugador puede elegir una carta o acción a comprar	
Pre-condición	• El Sistema debe estar en funcionamiento • El jugador está en su turno en la fase de acción/compra	
Post-condición	• Se permite o rechaza realizar la acción o compra	

Tabla 43. CU14 –Comprobar restantes

Identificador	CU 15	Comprobar tipo
Nombre	Comprobar tipo	
Actores	Usuario	
Objetivo	Comprobar que el jugador juega una carta del tipo correcto	
Curso normal	1. El sistema comprueba las cartas de la mano del jugador 2. El sistema muestra solo las del tipo correcto	
Pre-condición	• El Sistema debe estar en funcionamiento • El jugador está en su turno en la fase de acción/tesoros	
Post-condición	• Se permite o rechaza jugar la acción/tesoro	

Tabla 44. CU15 –Comprobar tipo

Identificador	CU 16	Jugar tesoros
Nombre	Jugar tesoros	
Actores	Usuario	
Objetivo	Jugar una o varias cartas de tesoro	
Curso normal		**Alternativas**
1. El sistema muestra por consola las opciones y cartas posibles 2. El jugador introduce por consola el identificador de la carta a jugar		a) 1.1 El sistema se salta la ronda porque el jugador no dispone de tesoros b) 2.1 El jugador introduce la opción de no jugar acciones c) 2.2 El jugador elige jugar todos los tesoros de una vez
Pre-condición	• El Sistema debe estar en funcionamiento • El jugador está en su turno en la fase de tesoros	
Post-condición	• La carta de tesoro de gestiona en el tablero para la ejecución de su lista de efectos	

Tabla 45. CU16 –Jugar Tesoros

Identificador	CU 17	Comprar cartas
Nombre	Comprar cartas	
Actores	Usuario	
Objetivo	Comprar una carta disponible de las pilas de suministros del tablero	
Curso normal		**Alternativas**
1. El sistema muestra las pilas que le jugador puede comprar y que no están vacías. 2. El jugador introduce por teclado el índex de la pila que quiere		a) 1.1 El sistema se salta la ronda porque el jugador no puede comprar ninguna carta b) 2.1 El jugador introduce la opción de no comprar cartas
Pre-condición	• El Sistema debe estar en funcionamiento • El jugador debe estar en la fase de compra • El jugador debe tener tesoros suficientes • El jugador debe tener compras suficientes • La pila debe tener cartas disponibles	
Post-condición	• La carta pasa a la pila de descartes del jugador • Se resta una carta de la pial de suministros elegida	

Tabla 46. CU17 –Salir del menú plantillas

Identificador	CU 18 Comprobar Compra
Nombre	Comprobar compra
Actores	Usuario
Objetivo	Comprobar que el jugador puede adquirir una carta de la pial de suministros
Curso normal	1. El sistema da a elegir la jugador únicamente pilas con cartas disponibles, que el jugador pueda hacer frente al pago y siempre que el jugador tenga compras disponibles
Pre-condición	• El Sistema debe estar en funcionamiento • El jugador debe estar en la fase de compra
Post-condición	• El sistema da una respuesta afirmativa o negativa al comprobar la pila dependiendo del cumplimiento de las 3 condiciones

Tabla 47. CU18 –Comprar cartas

3.4 Diseño

Una vez analizados los requisitos del juego y los casos de uso pasaremos a explicar cuál es el diseño del proyecto, adentrándonos en cada uno de los módulos principales de nuestra plataforma. A la hora de tomar decisiones de diseño, se han tenido en cuenta los requisitos expuestos anteriormente así como los posibles problemas encontrados al plantear los casos de uso. Durante la evolución del desarrollo, el diseño ha ido cambiando adaptándose a las necesidades que iban surgiendo así como a las mejoras de rendimiento, seguridad y eficacia. En el diseño se puede apreciar un modelo de diseño basado en el patrón Modelo vista controlador (MVC). Por el cual el motor u objeto modelo manda información a un objeto vista que recibe y muestra la información. Entonces el jugador o IA recibe la información e interactuando con un objeto Controlador transmite la información al motor. En nuestro caso la clase Inteligencia junto con la clase IOMenus actuarán en cierto modo como objetos Vista y Controlador dependiendo del tipo de usuario que controle el jugador.

3.4.1 Motor de juego

En este apartado haremos referencia al núcleo del juego, adjuntando el diagrama de clases que participa en el mismo. Nos centraremos en cómo gestiona la plataforma los diferentes estados del juego, tanto la creación y lanzamiento de la partida como la gestión que se hace de los jugadores, las cartas y los efectos de las mismas por parte de las diferentes clases del juego; así como en la organización de las diferentes clases, sus funciones y la comunicación entre ellas.

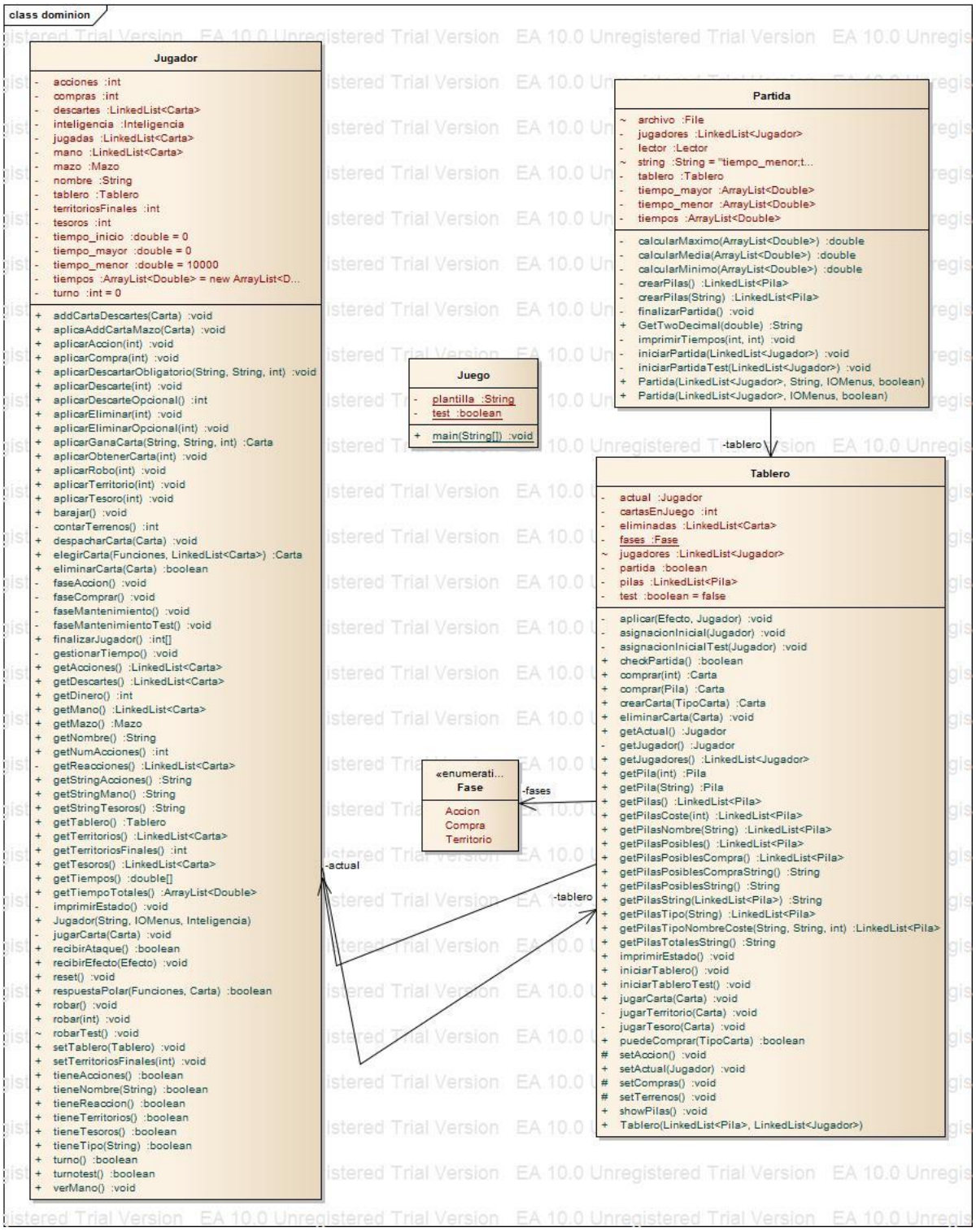


Ilustración 18. Paquete Dominion

Desde el punto de vista organizativo se ha intentado crear paquetes que incluyan el mínimo posible de clases para poder diferenciar claramente módulos funcionales. En este punto, analizaremos cada uno de los paquetes del programa y las clases que los forman adentrándonos en cada una de ellas y explicando sus funcionalidades; así como las razones por las cuales se han diseñado de esa forma. El diseño del diagrama de clases lo podemos ver en la ilustración 18.

3.4.1.1 Paquete Dominion

Es el paquete raíz que anida al resto de paquetes, tanto los que forman parte del motor como los que no. Las clases que podemos encontrar en este paquete son las directamente relativas a la mecánica del juego en sí y a la creación de partidas o simulación.

Juego: Es la clase "entrada" del programa. Es la que inicia el programa a través de la función main. Auto contiene los menús principales de partidas y la base de datos, es la puerta de entrada al simulador de Dominion.

- **Función principal:** Ser la clase de arranque del programa y además lanzar por consola el menú del simulador, desde el cual accedemos tanto a la gestión de partida como la de las plantillas cartas de juego.
- **Funcionalidades:**
  - Es la clase encargada de lanzar una partida, introduciendo por parámetro la lista de jugadores creados así como la ruta de la plantilla que se usará para definir las cartas que entran en juego y además del objeto IOMenus que posibilita la introducción de comandos así como las salidas por consola durante la partida.
  - Es capaz de listar las diferentes plantillas así como las cartas definidas en la base de datos que podrán ser añadidas en una plantilla.
  - Contiene los dos menús del juego, tanto el del simulador como el de la gestión de la base de datos (plantillas cartas).
  - Creación de los jugadores tanto de interacción humana como controlados por una IA.
  - Crea el objeto IOMenus que gestiona la entrada de comandos así como las salidas por consola de todo el programa. Este mismo objeto que será usado por la clase Juego, será introducido por parámetro en la Partida para que sea el medio de comunicación único y común a todos los objetos de la partida de Dominion.
  - Desde el menú de plantillas se pueden crear y editar archivos de plantilla que serán los que definan las pilas de suministros del juego.
  - Desde el menú plantillas se puede acceder a las cartas que existen, organizadas por tipos y, añadirlas a la plantilla que tengamos en el buffer.
  - Podemos añadir/editar/borrar cualquier carta de la plantilla así como el número de cartas que estarán disponibles en la partida de cada suministro por separado.

- **Decisiones de diseño:** El objetivo de esta clase fue externalizar lo máximo posible la creación de objetos de juego, de manera que pudieran gestionarse todos los objetos que toman parte en la misma sin necesidad de lanzar una partida en sí. Tanto el objeto de interfaces, como la clase Partida, el tablero y los jugadores que toman parte en la simulación de Dominion son creados desde la clase Juego. Además, la plantilla que se use en la partida se habrá seleccionado desde esta clase, así como editada o creada en caso de que así lo quisiese el usuario.

Partida

Esta clase se encarga de la gestión del inicio y fin de la partida. Es creada por la clase Juego y cuando se crea activa una serie de mecanismos para poder lanzar partidas independientes.

- **Función principal:** En gestión de partida nos referimos a la inicialización de objetos, asignación de turnos y el chequeo de la finalización del juego. En la ilustración 19 podemos ver claramente el proceso de creación de la clase partida y las clases que intervienen. Creando las pilas desde lector, inicializando el tablero y a la vez estableciendo el menú en la clase IOMenus.

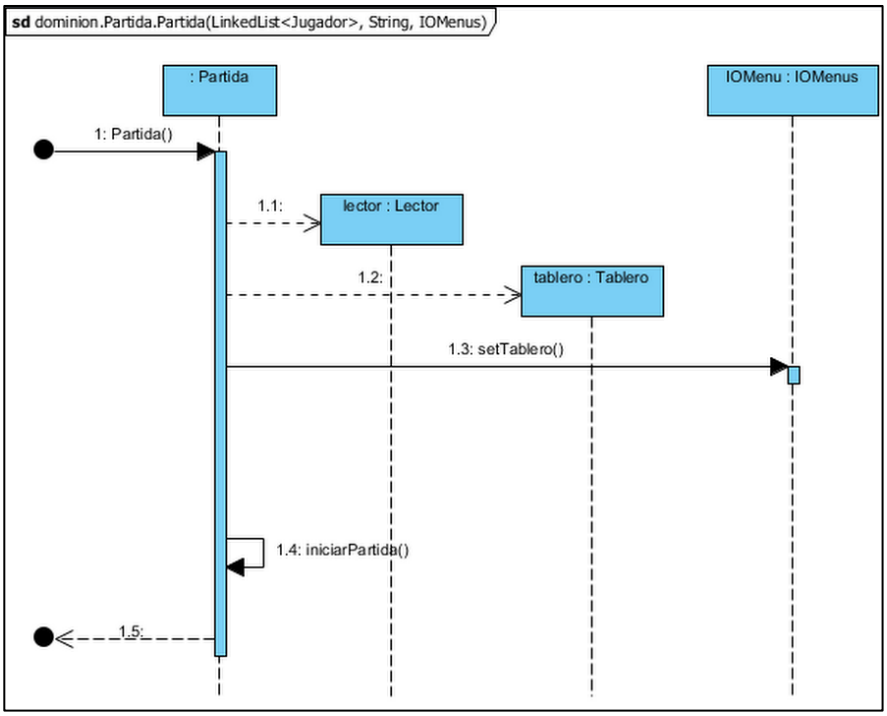


Ilustración 19. Proceso iniciado al crear Partida

- **Funcionalidades:**
  - Inicializar el tablero con los jugadores y las plantillas ya extraídas/creadas del fichero de plantillas.

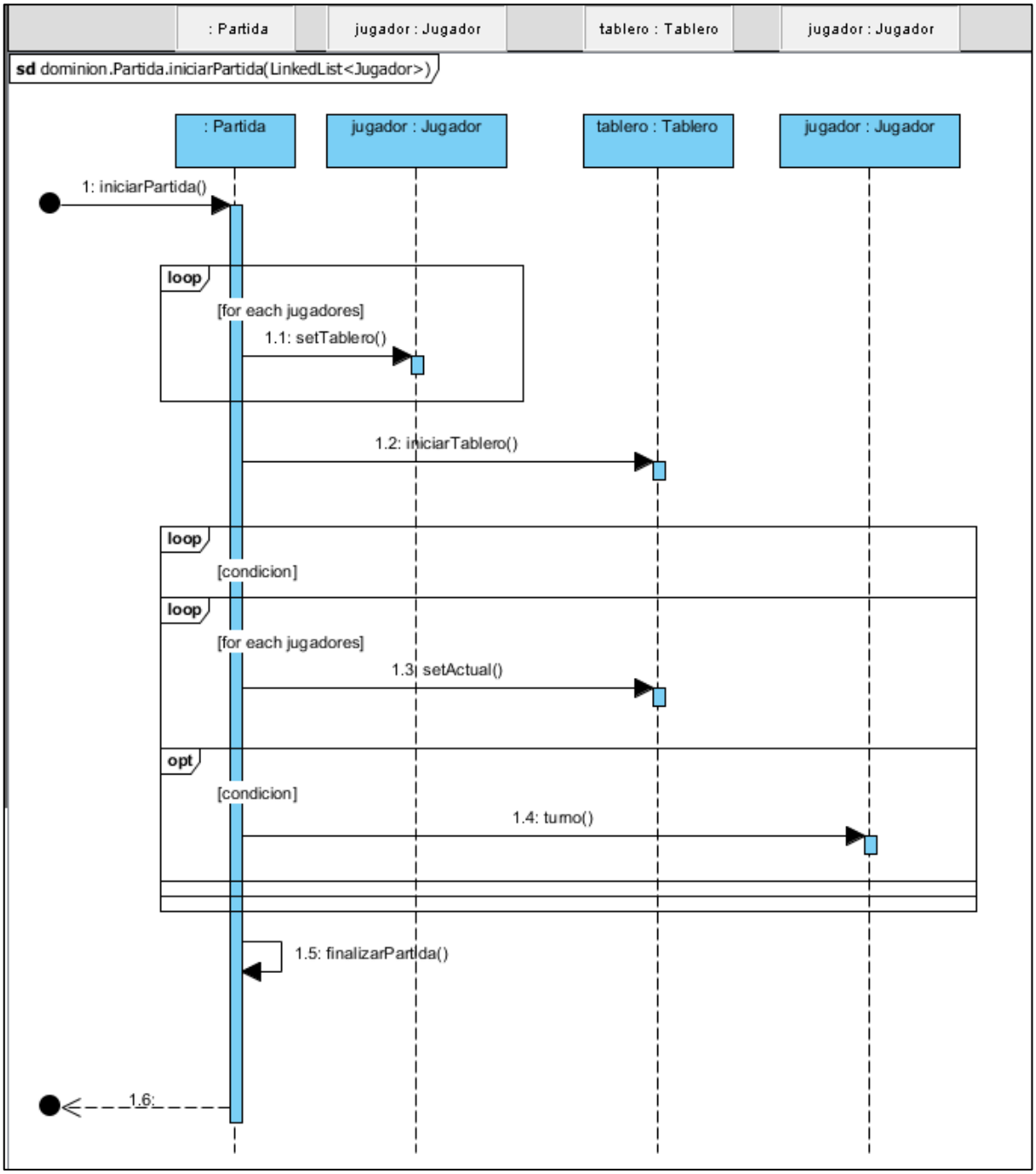


Ilustración 20. Inicialización de objetos de la partida

- Gestión de la partida: Podemos ver en la ilustración 20 cuál es el proceso por el cual desde que se llama a iniciarPartida() el sistema va estableciendo las condiciones iniciales, asignación de referencias, creación de pilas, etc. Y gestiona los turnos y chequea la finalización del juego.

- Gestión de turnos: En cada iteración se llama la función turno de un jugador, que es una función *boolean* que devuelve si la partida ha acabado o no. Siempre que se realiza un cambio de jugador, se establece dicho jugador como actual en el tablero de juego.
- Introducir en los jugadores la referencia al tablero de la partida.
- Finalizar la partida: Cuando la validación de condiciones al final de cada turno sea negativa y por lo tanto la partida esté finalizada.
- Contabilizar los puntos y turnos de cada jugador y mostrar por consola el orden de los jugadores.
- Cargar las pilas de suministros llamando a la clase Lector a partir de la plantilla de cartas, como se muestra en la ilustración 21. Y que se explicará con más detalle cuando analicemos la clase Lector.

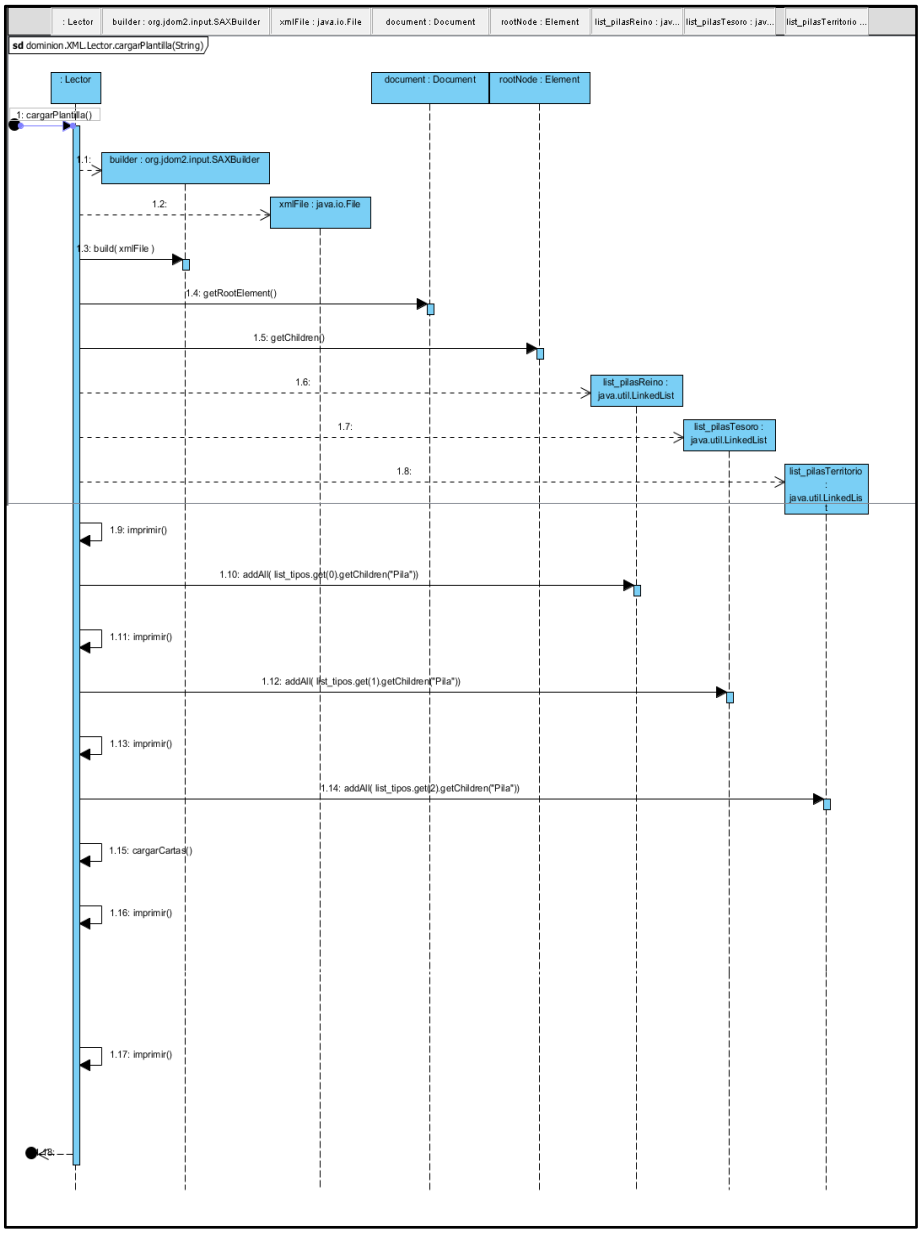


Ilustración 21. CargarPlantilla

- La inicialización del tablero comprende el reseteo de sus variables y sus listas así como asignar las cartas iniciales a los jugadores de la partida, como podemos ver en la ilustración 22. Se resetean todas las variables del tablero y luego se asignan las cartas de inicio a cada jugador de manera que cada partida se inicie de manera *"limpia".*

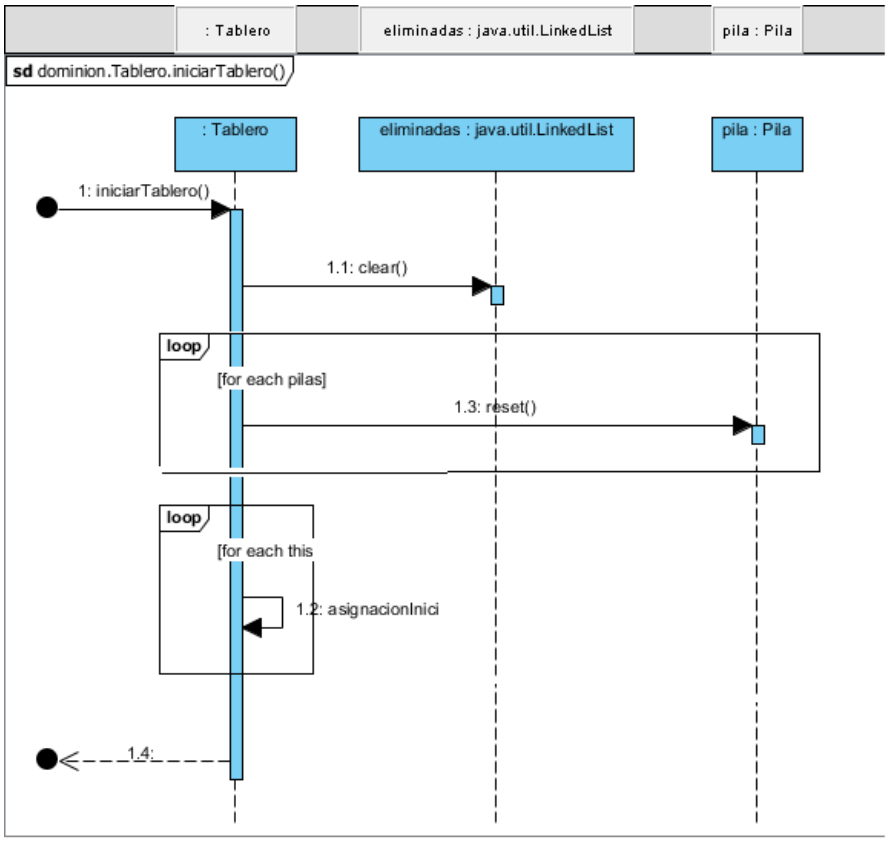


Ilustración 22. Inicializar tablero

- **Decisiones de diseño:** Desde un punto de vista organizativo podemos decir que la clase partida representa la partida real, sin más funciones u adornos, guardando una referencia al tablero, a la interfaz y a la lista de jugadores que toman parte en la partida. Deposita toda la gestión de la partida en la clase tablero evitando que los jugadores o el mismo tablero interactúen con la clase partida de manera que la gestión de turnos y lanzamiento o finalización de partidas sea un proceso totalmente automatizado y externo a la mecánica del juego. Esta independencia en el proceso de inicialización, creación y posterior reseteo de objetos, al finalizar cada partida unitaria, queda patente en el hecho de que si llamásemos al método CrearPartida() desde dentro de un bucle, se podrían lanzar varias partidas de manera secuencial totalmente independientes con los mismos jugadores, plantilla y tablero. A destacar la gestión de los turnos configurando la función turno como un *boolean* siendo la devolución del mismo el condicionante para seguir con las iteraciones.

Tablero

Es la clase central de la mecánica de la partida, gestionando la obtención de cartas así como la gestión de los efectos de las mismas cuando las pone en juego uno de los jugadores.

- **Funcionalidad principal:** Gestión de las cartas del juego y del desarrollo de las diferentes fases de cada turno.
- **Funcionalidades:**
  - Adquisición de cartas: Cuando un jugador quiere obtener una carta de un suministro ya sea mediante una compra o una obtención de cartas, se llama a tablero indicando el tipo de carta o el índice absoluto del suministro en la lista de suministros del tablero. Además se ha definido una función para saber si un jugador puede comprar una carta comparando el dinero que tiene junto con el que cuesta dicha carta. De cada pila se obtiene el modelo de la carta y posteriormente se crea la carta concreta pasando por parámetro dicho modelo.
  - Obtener suministros: Se han diseñado varias funciones para obtener la lista de pilas de suministros además de para mostrar por pantalla dicha lista y su correspondiente índice relativo. Un jugador accederá a las pilas por dos razones: o para comprar o para obtener una carta por el efecto de una carta de acción. Por ello, deberemos dar la posibilidad de obtener las pilas de suministros posibles en su totalidad o solo las posibles en caso de compra, es decir, que cuesten igual o menos tesoros de los que posee el jugador. En el caso de que queramos obtener las pilas en su totalidad pero de un solo tipo, nombre o con un coste máximo también se han definido las correspondientes funciones.
  - Comprobar partida: Se ha diseñado una función que comprueba cuántas pilas están vacías y si se ha acabado el suministro de Provincias para saber si la partida ha acabado o sigue en funcionamiento.

- JugarCarta: Siempre que un jugador juegue una carta tiene que hacerlo invocando al tablero y pasando la carta por parámetro. El tablero leerá una lista de efectos u otra de la carta dependiendo del momento del turno que sea (Acciones, Compra, Terrenos). Una vez obtenida la lista de efectos independientes se gestionarán uno por uno de manera independiente leyendo del mismo efecto a qué jugadores ha de aplicarse, "Actual, Resto o Todos" definido en el enumerado Objetivos. Este proceso se puede ver en la ilustración 23.

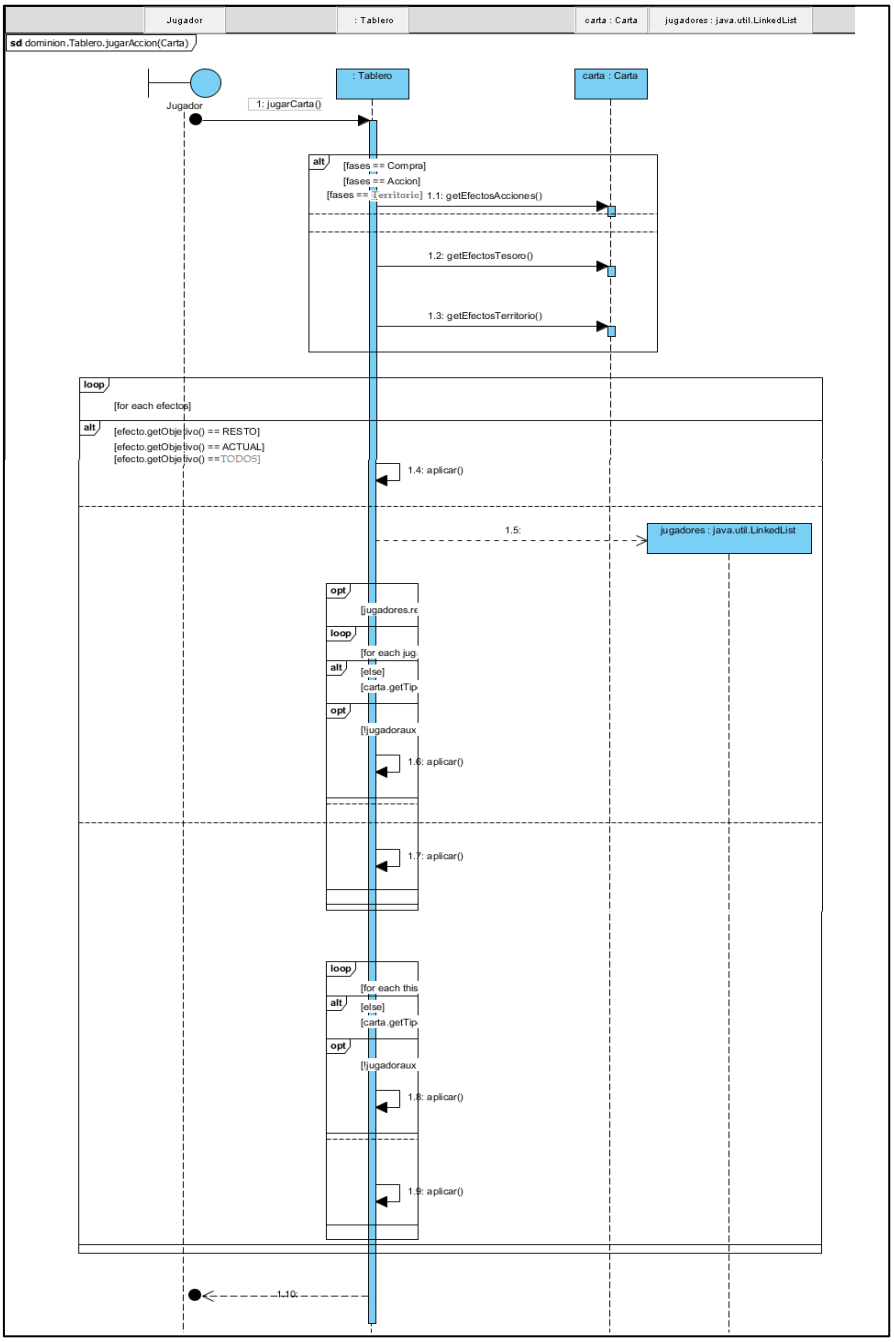


Ilustración 23. Tablero.JugarCarta

- **Decisiones de diseño:** La clase Tablero guarda referencias de prácticamente todos los objetos que toman parte en la partida. Es el eje central del juego el cual almacena todas las cartas eliminadas y pilas de suministros. Además de tener la lista de jugadores, siendo el nexo de unión que permite no solo usar cartas que involucren a más jugadores aparte del propietario de la carta sino que; además gestiona que una posible inteligencia artificial no pueda efectuar operaciones sobre el resto de jugadores de forma desautorizada. Cada vez que obtenemos una carta del tablero por una acción o una compra esta se devuelve tal cual sin añadirla a ninguna de las listas del jugador, debido a los diferentes destinos que puede tener dicha carta y para dejar la libertad de implementar dichos destinos.

Jugador

Dentro del paquete Dominion, el jugador es en cierta manera la carcasa del jugador. Define las propiedades que tendría un jugador de Dominion así como pone a disposición del usuario o de la inteligencia artificial todo el set de funciones que necesita para interactuar en el juego.

- **Funcionalidad principal:** Ofrecer todas las operaciones y atributos que un jugador tendría.
- **Funcionalidades:**
  - Comprar cartas: El sistema comprueba que el jugador tenga compras restantes siempre antes de ofrecer la posibilidad de adquirir cartas, saltando a la siguiente fase en caso negativo.
  - Jugar cartas: El jugador tendrá que jugar las cartas siempre en su turno y en la fase del mismo que indique el tipo de la carta. El jugador dispondrá de varios momentos para jugar cartas, la de tesoro la de acción o la de jugar foso para evitar ataques. El sistema saltará de fase si no se dispone de cartas del tipo correcto. Además, que el jugador solo podrá seleccionar cartas de ese tipo ya que será las que se indiquen al tablero por parámetro a la hora de obtener las pilas seleccionables. Hay varias formas de despachar la carta, puede ir a la pila de cartas jugadas o puede eliminarse tras su uso. El proceso se ve más claramente en la ilustración 23.

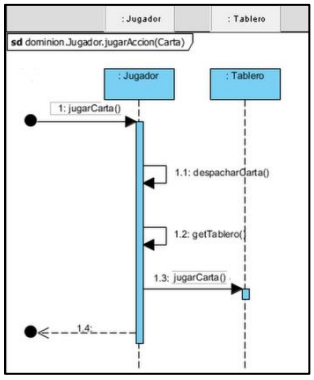


Ilustración 24. Jugador. Jugar carta

- Jugar Acción: Se jugaran tantas acciones como se tengan, se brinda la posibilidad de saltar el turno.
- Jugar Tesoros: La interfaz además de dar la posibilidad de jugar los tesoros uno por uno o saltar ronda también nos permite jugar todos los tesoros de una vez.
- Fase mantenimiento: Se añaden todas las cartas de la mano y de la lista de jugadas al mazo de descartes y se limpian dichas listas, además se roban las cartas para el siguiente turno.
- Robar: Si es el primer turno se roban cartas cuando se invoca a turno() desde Partida, en caso contrario siempre se roba la final de la fase de mantenimiento. Se ha configurado un método para robar un número de cartas indicado por parámetro que pueda dar soporte a los efectos de acción que obligan o dan la opción de robar cartas.
- Aplicación de efectos: En esta clase encontraremos diferentes funciones con la nomenclatura aplicar*() así como otra serie de métodos que se usan para el normal desarrollo del turno. Estas funciones se invocarán desde las cartas del juego cuando el tablero gestione los efectos de la carta cuando se pone en juego. Destacar el efecto de obtención de cartas que recibe 3 parámetros y es capaz de filtrar pilas con respecto a esos tres parámetros devolviendo una lista de pilas que únicamente cumpla dichos parámetros.

- **Decisiones de diseño:** Entre los atributos podemos nombrar la mano del jugador, las cartas que tiene en el mazo, las descartadas, así como los turnos jugadores, las acciones y compras restantes, los tesoros que posee, etc…. Además, ofrece el set entero de funciones para recibir efectos. Dichas funciones son llamadas por las propias cartas una vez son invocadas desde el tablero. Esta clase es totalmente independiente de lo que será su inteligencia o controlador, del cual guarda una referencia introducida por parámetro cuando se crea un objeto jugador.

3.4.1.2 Efectos

Este paquete almacena todas las clases Efecto del juego, que están diseñados de manera que todos hereden de la clase Efecto y, que en cada uno se re-implemente los métodos tanto de construcción como de efecto que tienen en la partida.

Efecto

Cada efecto es independiente de cualquier otro y por defecto se crea indicando por parámetro la cantidad, ya que muchos efectos se pueden medir en términos cuantitativos además de indicar también los objetivos. Para algunos casos concretos se crea un nuevo constructor añadiendo un parámetro String que indica un tipo o nombre concreto necesario para la aplicación del efecto.

- **Decisiones de diseño:** Se han diseñado los efectos de manera que funcionen de una manera similar al patrón Visitor. Se implementan clases visitantes que serán los efectos y los visitados son los jugadores. La diferencia es que la función de aceptar el visitante podemos considerar que invoca tablero al ejecutar los efectos o el propio jugador al jugar la carta. El atributo cantidad se dejó como protected para que las clases que heredan dentro del paquete pudieran manejarlo a placer. El diagrama de clases podemos verlo en la ilustración 25.

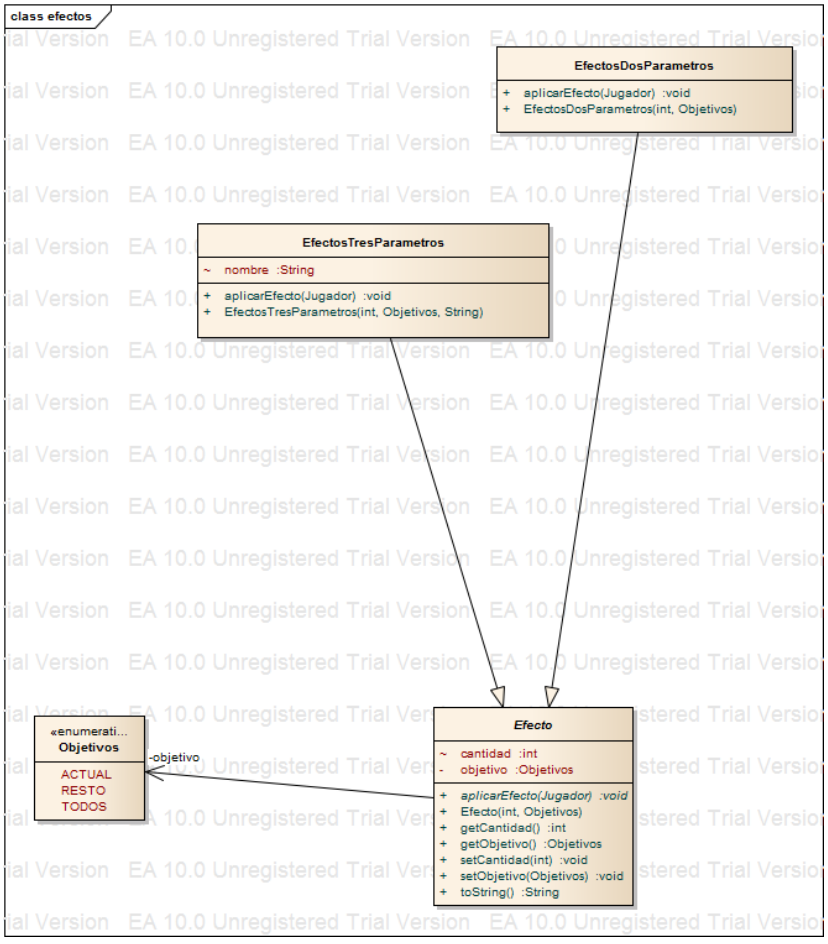


Ilustración 25. Paquete Efectos

3.4.1.3 Cartas

Este paquete engloba a todas las clases que están relacionadas con las cartas o su gestión. Aquí incluimos el mazo, las pilas de suministro, la clase de la carta así como la de su tipo, al igual que incluimos la clase tipos que será la clase auxiliar que ayuda a crear las cartas a partir de la base de datos. En la ilustración 26 se muestra el diagrama de clases

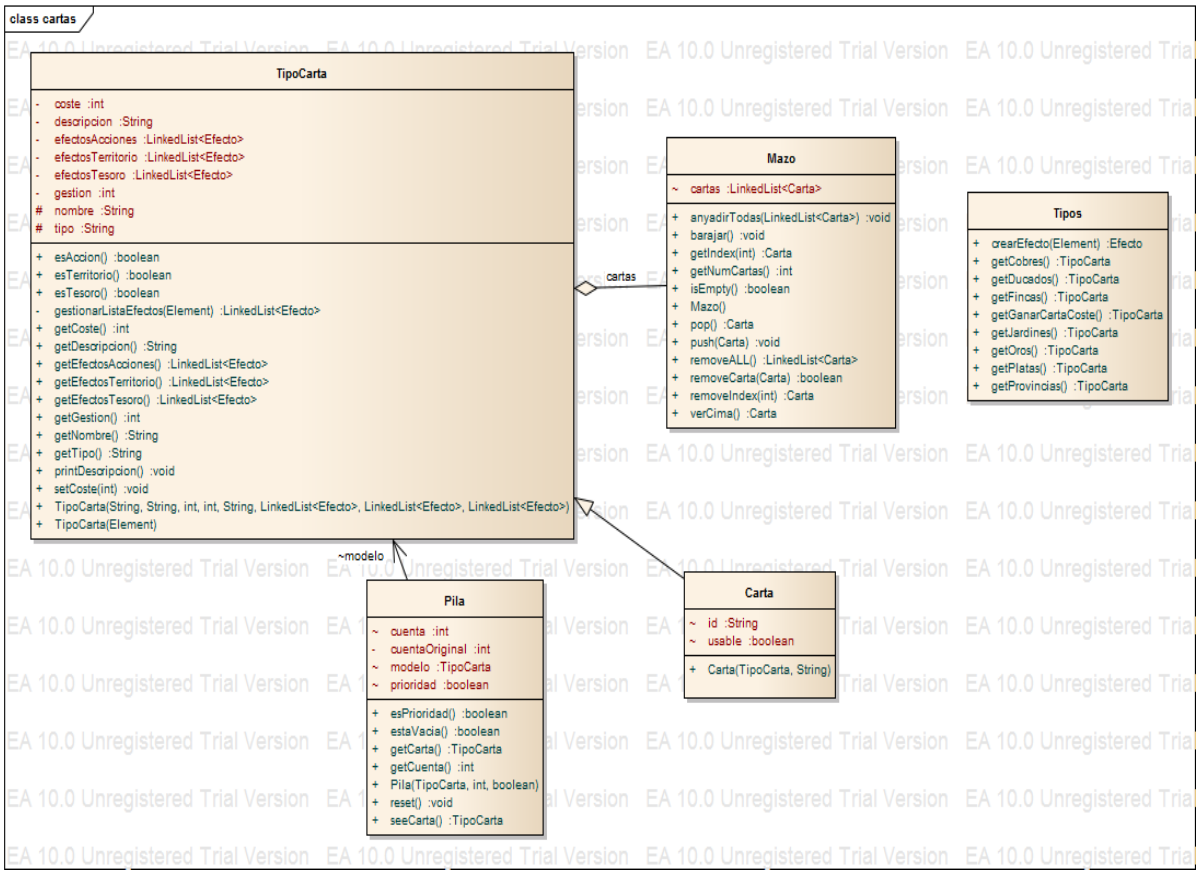


Ilustración 26. Paquete Cartas

TipoCarta

Define el modelo que seguirán las cartas. Entre sus atributos encontramos el nombre y tipo de la carta además del coste en tesoros. Por otro lado, contiene las listas de efectos que definen cuál será su repercusión en el juego.

- **Funcionalidad principal:** Encapsular una lista de efectos que realicen diferentes acciones en la partida y sus jugadores.
- **Funcionalidades:**
  - Listas de efectos: Hay 3 fases diferenciadas del juego donde se juegan cartas que además son efectos y cartas de tipos concretos. Se ha creado una lista para cada una de esas fases de manera que en cada fase el juego cargue y ejecute la lista de efectos que corresponda.

- Tipo: Hemos definido 5 tipos para las cartas, tesoro, territorio, acción, reacción y ataque. Cada una se juega en un momento diferente de la partida, para eso se han implementado varios métodos en TipoCarta que actúan como predicados para comprobar si podemos jugar la carta en ese momento.

- **Decisiones de diseño:** Se crearon las listas de efectos para que el tablero pudiera leerlas en el orden específico que queramos y dar la posibilidad de analizar cada efecto por separado. Esta manera de identificar o definir las cartas nos da la posibilidad de crear cartas a placer, ya que una carta se diferencia de otra por los efectos que contiene en cada una de sus listas. Gracias a esto podemos cubrir futuras ampliaciones o extensiones del juego , así como cambios más profundos gracias a que el elemento básico de este tipo de juego, que son las cartas, están definidas por efectos independientes que podemos crear y editar fácilmente y así cumplimos uno de los objetivos que nos marcamos con el proyecto. A la vez que se decidió crear un constructor que decodificara los elementos XML para descargar la clase lector, se decidió incluir en esta clase la gestión de la de las listas de efectos, imprimiendo por pantalla las propiedades de cada efecto decodificado y mandando a la clase tipo los efectos individuales codificados para su creación, quedando TipoCarta con la función de añadirlos todos a una lista una vez creados asignándolos, desde el constructor al atributo adecuado.

Mazo

Es el objeto que almacena las cartas que posee el jugador.

- **Funcionalidad principal:** Gestionar las cartas del jugador y ofrecer las operaciones necesarias de introducción, extracción así como visualización de las mismas.
  - Extracción: Esta clase da la posibilidad de extraer las cartas de la cima como si fuera una pila o de extraer las cartas introduciendo su posición en el mazo.
  - Barajar: Es una de las razones por las que se creó esta clase contenedora y es que en Dominion cuando se acaban las cartas para robar se vuelven a introducir todas en el mazo y se han de barajar. Se ha utilizado una función de *Collections* para ello.
  - Visualizar: Para algunos efectos es necesario visualizar algunas cartas sin por ello sacarlas del mazo.

- Decisiones de diseño: Si bien se podrían haber usado las clases contenedoras List o *LinkedList* directamente se prefirió crear esta clase contenedora específica por cuestiones de organización y limpieza en el código.

Pila

Esta clase representa las pilas de suministros de cartas del juego. Tienen un tipo de carta almacenado y una cuenta de cartas disponibles.

- **Funcionalidad principal:** Almacenar el tipo de carta que estará disponible en el juego regulando el número de veces que puede aparecer en el mismo.
- **Funcionalidades:**
  - Extracción: Cada vez que se extrae el modelo para crear una carta se resta en una unidad las copias disponibles del TipoCarta, de esta manera controlamos las copias que hay en juego así como las unidades restantes para posibles decisiones de la IA. De la misma manera se podrá evaluar si la partida ha terminado o no. Cabe recordar que una partida acaba si la pila de la carta *"Provincia"* se acaba o 3 suministros cualquiera se acaban.
  - Visualización: Para evaluar decisiones de compra, adquisición o simplemente hacer comprobaciones de compra se necesitará acceder a las características del modelo sin que eso signifique que vamos a crear una carta.
  - Prioridad: Siempre que una pila prioritaria se acabe terminara la partida. En el diseño original de Dominion la única pila prioritaria es la de "Provincia" pero se ha dejado abierta la posibilidad para posibles ampliaciones o cambios.
- Decisiones de diseño: Esta clase también se creó por cuestiones de organización y elegancia en el código, además de que facilita la gestión de cartas sin olvidar que con este diseño ahorramos espacio creando solo los objetos cuando es necesario. Existe un atributo cuentaOriginal que se establece desde la construcción del objeto que guarda e máximo original para cuando se resetean las pilas.

Carta

Podemos decir que en cierta manera es una instanciación de TipoCarta encapsulado en otra clase.

- **Funcionalidad principal:** Como elemento básico del juego su funcionalidad es en cierta manera, la misma que la de TipoCarta. Pero, si TipoCarta es como la definición abstracta, la carta será el objeto palpable o físico con un ID asignado que formará parte de la mecánica del juego.
- **Decisiones de diseño:** Se creó esta clase para tener pilas con un diseño modelo y que las cartas fueran ejemplos de tales modelos pero con un id asignado que representa su número dentro de la partida. En cierta manera podemos diferenciar TipoCarta como la identidad de la carta y Carta como su representación en el juego, la parte dinámica. Todos los atributos que no estén directamente relacionados con la identidad de la carta si no con su estado en la partida, como su Id, si es usable y futuras ampliaciones, irán definidos en esta clase.

Tipos

Esta clase está diseñada para ser invocada por la clase Lector que gestiona la base de datos y sirve para convertir en efectos Java los efectos definidos por cada carta en la base de datos.

- **Funcionalidad principal:** Recibir por parámetro desde la clase Lector la lista de efectos codificada en el XML y, crear en Java sus homónimos extrayendo sus parámetros y llamando al constructor correspondiente.
- **Funcionalidades:** Además de poder servir de conversor de XML a objetos Java, podemos usarlo para crear pilas, creando métodos específicos para cada carta e invocándolos directamente desde algún método de la clase Partida.
- **Decisiones de diseño:** Se ha intentado externalizar lo máximo posible la base de datos del juego pero aun así, los efectos deben tener una codificación en Java. Entre todas las posibilidades se decidió escribir directamente el nombre del efecto en la base de datos, en vez de codificarlo como una lista de números ya que el cambio no era muy grande en cuestión de espacio o líneas de código; pero si en cuestión de legibilidad tanto de código como directamente la base de datos.

3.4.1.4 Duplas

Este paquete contiene 3 clases que son totalmente auxiliares y sirven para mostrar correctamente el orden en el que quedan los jugadores al acabar cada partida. En general, se decidió usar este paquete con vistas a un futuro en el cual la victoria siga otros criterios diferentes y así, poder regular dichos criterios sin tener que tocar el resto del motor. Y del cual podemos ver su diagrama de clases en la ilustración 27.

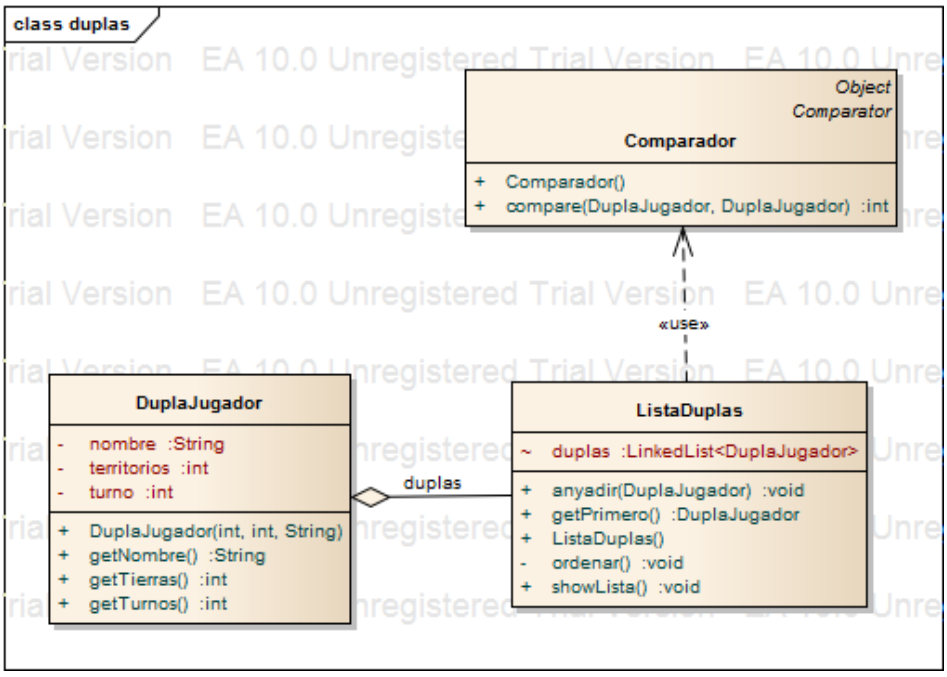


Ilustración 27. Paquete Duplas

Comparador:

Es una clase que compara objetos de la clase DuplaJugador.

- **Funcionalidad principal:** Compara los diferentes jugadores para poder ordenarlos con respecto a sus puntos de victoria y turnos jugados.
  - **Funcionalidades:**
    - Comparator: Re-escribiremos un método de comparación que devuelve un número positivo siempre que el primer objeto a analizar sea mayor que el segundo y un número negativo si es al contrario. En caso de ser iguales devolverá 0.
- **Decisiones de diseño:** Esta clase implementa la interfaz Comparator<Object> que ofrece un set de funciones de comparación para objetos genéricos.

DuplaJugador

Es una clase que encapsula a un jugador y que sirve para realizar el orden final de victoria.

- **Funcionalidad principal:** Recoger los datos que definen la victoria de un jugador concreto.
- **Funciones:**
  - Obtener tierras o turnos: Al representar a un jugador de Dominion en la comparación final de resultados, estas funciones al ser invocadas devuelven los puntos de victoria o los turnos jugados por cada uno.
- **Decisiones de diseño:** Se optó por implementar esta clase para extraer del jugador únicamente los datos que nos interesasen dejando intacta la clase de origen.

ListaDuplas

Es una clase contenedora que usa un *LinkedList* para almacenar objetos JugadorDupla.

- **Funcionalidad principal:** Siempre que se añade un jugador ordena la lista con respecto al comparador implementado.
- **Funcionalidades:**
  - Añadir: Siempre que se añade un elemento se inserta al principio de la lista y se invoca al método interno de ordenación.
  - Comparar: Se persigue que cada vez que se inserta un elemento nuevo la lista quede ordenada.
- **Decisiones de diseño:** Se decidió implementar una lista nueva para tener el pack auxiliar completo, el comparador, el elemento y por supuesto el contenedor.

3.4.2 Interfaz IA/Humano

Tomando en referencia el resto de proyectos basados en Dominion y a uno de los objetivos marcados por el tutor, se marcó como objetivo externalizar totalmente la interfaz de usuario tanto para máquinas como para personas del motor del juego, convirtiéndose este en una caja negra que emite preguntas y ofrece ciertas funciones a las inteligencias tanto humanas como artificiales.

A la vez que explicaremos dicho diseño, también analizaremos por separado cómo se ha planificado la IA. La idea es alejarnos un poco del método del simulador Geronimoo o del Dominion Simulator, que funcionan con reglas de condiciones y cartas con prioridades asignadas como unidad.

Diseño Interfaces

En este apartado haremos un análisis específico de cómo se ha diseñado el sistema en lo referente a interactuar con el usuario así como al diseño del control del jugador. El objetivo era que se pudieran tratar indistintamente jugadores humanos o jugadores controlados por inteligencias, que tuvieran el mismo acceso al motor y que además, estuvieran definidos de manera muy clara para facilitar su modificación y extensión posterior sin por ello, tener que modificar el resto del código.

- Interfaces: En este paquete encontraremos un enumerado que definirá las preguntas a las que debe responder cualquier jugador durante la partida así como las clases que implementan Inteligencia, que es una clase abstracta que define la gama de funciones que englobaran las respuestas específicas. Además, la clase que posibilita la entrada de valores por teclado y la impresión por consola también se ha añadido en este paquete como parte funcional de la interfaz. El diagrama de clases podemos verlo en la ilustración 28.

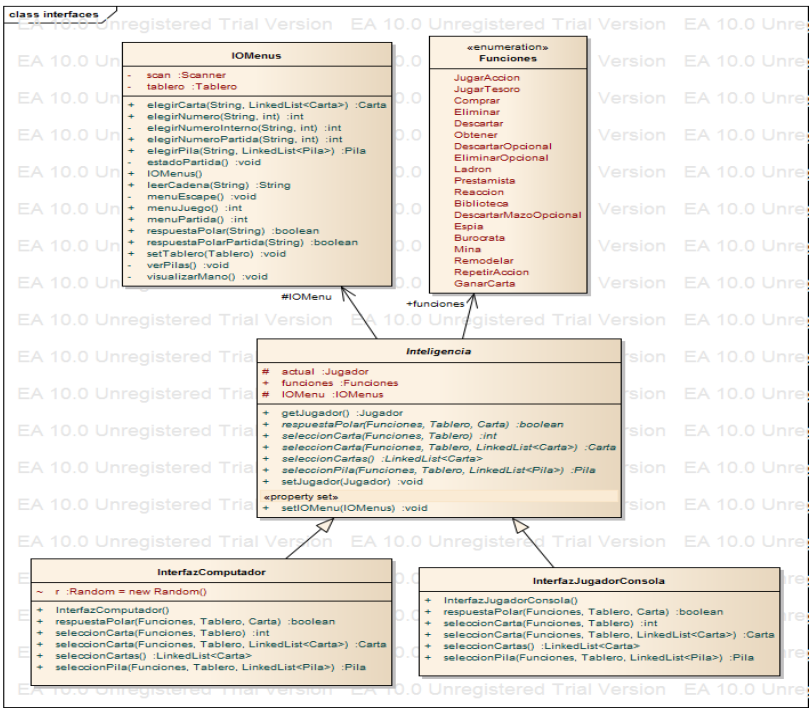


Ilustración 28. Paquete Interfaces

Funciones

Es una clase que define una serie de enumerados que definen las respuestas esperadas por el sistema.

- **Funcionalidad principal:** Cada uno de estos enumerados dota de semántica a la respuesta.
- **Decisiones de diseño:** En vez de implementar varias funciones con los mismos parámetros y con los mismos valores de retorno, por cuestiones de elegancia en el código y por facilidad a la hora de posibles extensiones se decidió utilizar enumerados.

Inteligencia

Es la interfaz que define el marco de las clases que implementarán las inteligencias de los jugadores.

- **Funcionalidad principal:** Definir los métodos que tienen que implementar las clases que heredan. A la hora de crear una inteligencia, esto es necesario para que tanto la clase que creemos para humanos como las subsiguientes inteligencias artificiales respondan exactamente a las mismas funciones/preguntas.
- **Funcionalidades:**
  - Selección de Carta: Existe de dos tipos. Una selección sobre una lista que se pasa por parámetro y que devolverá una carta de las componentes de la lista y otra selección que obtenga una lista del objeto tablero y que devuelva un entero con el índex de la carta seleccionada.
  - Seleccionar Pila: Devuelve una pila sobre una lista de pilas posibles que se pasa por parámetro.
  - Seleccionar Cartas: Permite que el jugador pueda devolver una lista de cartas en caso de que el sistema lo pida o necesite.
  - Respuesta polar: Hay preguntas del sistema que necesitan un sí o no, este método da cabida a dichas consultas.
- **Decisiones de diseño:**
  - Conseguir externalizar toda la parte interactiva del sistema no fue un trabajo sencillo ya que todo estaba integrado en la misma mecánica del juego, como un primer avance para conseguir un sistema funcional. En un primer paso se extrajeron las peticiones por consola al usuario a una clase externa, que es IOMenus.
  - El siguiente paso fue extender dicha modificación a las clases que heredan de Efecto externalizando totalmente todas las decisiones interactivas.
  - Como último paso, hubo que poner otro punto intermedio en la toma de dichas decisiones creando la interfaz inteligencia y poniendo como dicho nexo las implementaciones de la interfaz; sabiendo a las funciones que tenía que responder el usuario para desarrollar una partida con cualquiera de los efectos existentes.

- Además, a cada una de esas funciones se la dotó de una semántica para poder ser más específicos en las respuestas y darle una mayor potencia a la IA.

IOMenus

Esta clase es la que posibilita la interactuación entre un usuario y el juego.

- **Funcionalidad principal**: Siempre que se necesite una respuesta por parte del usuario, su objeto inteligencia llamará a esta clase para que sea el humano el que mediante la introducción de comandos por consola tome las decisiones.
- **Funcionalidades:**
  - Menu juego: Es la primera función invocada del juego y muestra por pantalla las opciones principales del menú juego a la vez que recoge la decisión del usuario.
  - Elección de número: Se usa tanto de forma interna como de forma externa. Sirve para que el usuario pueda elegir entre un rango de opciones limitado por parámetro. Existen dos funciones que realizan la misma operación, una de ellas incluye un acceso al menú auxiliar de juego que ahora explicaremos.
  - Menú Auxiliar: Se decidió habilitar un menú que durante la partida permitiera consultar información de interés para el usuario como su propia mano, el estado de la partida o consultar las pilas de suministros y sus propiedades.
  - Elección de Carta o Pila: Sobre una lista de cartas o pilas que además se muestra por pantalla, se pide que se seleccione una que será el valor de retorno del método.
  - RespuestaPolar: En caso de que el sistema necesite una respuesta afirmativa o negativa del usuario, será esta función la encargada de mostrar las opciones por pantalla; así como de transformar el valor introducido en un valor booleano.
- **Decisiones de diseño:** Mientras que las respuestas que pida el sistema serán gestionadas por una IA a través de la clase InterfazComputador donde se implementará dicha inteligencia, la clase IOMenus dará soporte a la inteligencia humana definida en la clase InterfazJugadorConsola consiguiendo centralizar toda la interactuación entre ordenador y el usuario. Se decidió mantener una referencia a tablero para siempre tener acceso al estado real de la partida.

InterfazComputador

Esta clase implementa a Inteligencia y define las funciones que una IA debería responder para poder jugar a Dominion sin ningún problema o carencia.

- **Decisiones de diseño:** Se ha realizado el esqueleto de la IA pero no se ha implementado en detalle una inteligencia que pueda considerarse "inteligente" ya que no es el objetivo del proyecto. Aun así, y para poder hacer pruebas de funcionamiento, se han definido ciertas funciones para que la máquina siempre juegue todos los tesoros o compre cartas, a la vez que siempre que pueda elegir no descartarse lo haga.

InterfazJugadorConsola

Esta clase es la que representa la inteligencia humana en el juego, pidiendo y recibiendo indicaciones por consola siempre que sea invocada desde el motor del juego.

- **Decisiones de diseño:** Como dijimos antes, si bien en un principio la clase IOMenus era la responsable de toda la entrada y salida de comandos, en el momento en el que se homogeneizó la interacción tanto de usuarios como de IAs añadiendo un objeto inteligencia en cada jugador; esta clase pasó a ser la que recibiera las llamadas desde el motor del juego. Por lo tanto, debía tener la referencia tanto a IOMenus como al tablero de juego y, estar capacitada para mostrar al usuario todas las opciones que disponía así como el contexto real de cada pregunta o decisión.

Diseño de la IA

Al igual que con el resto de módulos del sistema, en cierta manera, se pensó en crear una IA o en dar la posibilidad de que la IA fuera totalmente externa al sistema; de manera que no estuviera recogida dentro del funcionamiento interno del motor.

Externalización de respuestas.

Gracias a nuestro diseño cada respuesta que se da al sistema o cada pregunta que hace el mismo a la Inteligencia, humana o computacional, está totalmente acotada y contextualizada. Hay una pregunta específica para cada acción del juego. No es lo mismo elegir una carta debido al Efecto Ladrón que debido al Efecto Prestamista por ejemplo, ya que las repercusiones serán totalmente diferentes. Aun así, el diseñador de la IA siempre tendrá la posibilidad de tratar dichas respuestas de la misma manera.

Creemos que este diseño facilita enormemente la labor de diseñadores y desarrolladores ya que no solo saben a qué funciones tienen que responder, simplemente creando una clase que implemente la interfaz "Inteligencia" si no que además, puede ir más allá y gracias al enumerado Funciones, puede contextualizar cada una de sus respuestas. Sin olvidar que, dejando abierta dicha posibilidad de herencia podemos diseñar IAs que sigan los algoritmos elegidos a placer, sin por ello interferir en la mecánica del juego o el resto de clases.

Evaluación de cartas

Tras analizar los sistemas de IA, tanto implementados como las opciones de ampliación, de las otras soluciones de Dominion podemos decir que nuestro diseño ofrece una mayor flexibilidad no solo a la hora de la construcción sino también a la hora de evaluar decisiones. Decimos esto porque si nos fijamos bien, en los anteriores cada carta llevaba asignada una prioridad que no necesariamente era totalmente acorde a la realidad de la carta ya que, se asignan de manera totalmente arbitraria en la creación de las mismas, siendo dicha prioridad una propiedad encapsulada en la carta que da poco lugar a ampliaciones o modificaciones.

Lo que se propone aquí es un salto en cuestiones de diseño tanto de cartas como de IAs, una carta deja de ser un elemento unitario y opaco para pasar a ser una lista de efectos que el propio programador añade, edita o borra. Con esto no solo se consigue flexibilidad en la creación de cartas, si no en su propia evaluación que pasará a ser parte del algoritmo externo diseñado para cada IA concreta y no interno de cada carta creada con anterioridad.

Se puede conseguir que las decisiones sean evaluadas por los efectos que realizan, siendo analizado cada uno de ellos de manera unitaria, evitando decisiones arbitrarias y consiguiendo evaluaciones reales de la repercusión de dicha carta en el juego. Por ejemplo, en caso de que decidiéramos crear una IA que funcionara por evaluación de efectos, con solo evaluar los efectos ya no solo tendríamos evaluadas todas las cartas sino además futuras modificaciones o creaciones. Esos más, la evaluación no tendría por qué ser rígida o asignar a un valor estático si no que, al depender del diseñador y no de la estructura ya creada se pueden diseñar tantos algoritmos diferentes como se quiera.

3.4.3 Base de datos XML

A la hora de decidir qué base de datos usar, tanto el lenguaje o paradigma así como la plataforma de gestión se tuvo mucho en cuenta la facilidad de implementación en Java o la existencia de librerías que facilitaran el trabajo. Las dos opciones rápidamente nos vienen a la mente son SQL y XML.

- SQL: Quizás el estándar más generalizado en lo que a bases de datos se refiere. Es un lenguaje que posibilita la creación de bases de datos relacionales. Tan generalizado que podemos encontrar muchas librerías y programas que ayuden a gestionar dichas bases de datos. Está preparado para realizar consultas complejas a la vez que extensas, tras más de 20 años de historia podemos decir que SQL es el lenguaje más robusto consistente y que ofrece los productos más útiles del mercado. Permite crear grandes bases de datos con un gran número de conceptos y atributos por concepto.
- XML: Es un lenguaje de código libre que nació con el fin de mejorar el intercambio de información en la red, proveyendo de un código que permitiera estructurar información y que fuese legible en varias plataformas. Además, su definición permite una fácil interacción del usuario con los ficheros sin necesidad de un software específico de gestión.

Al final nos decidimos por XML, lo primero porque es completamente libre y no necesita de licencias de ningún tipo. Además, permite la interacción directa del usuario con los ficheros debido a su bajo nivel de complejidad, no solo del contenido de los mismos si no del lenguaje XML en sí mismo. Su integración en Java es bastante sencilla y mucho más en ficheros de “peso ligero” como los nuestros. Consideramos que mediante varias etiquetas, una estructura de árbol y unos atributos bien escogidos podríamos dar una buena cobertura al programa en lo que a información persistente se refiere.

Para contener todos los datos referentes al juego se optó por usar 2 ficheros diferentes que más adelante analizaremos en profundidad:

- Cartas: Este fichero corresponde al repositorio de cartas que podrán usarse en el juego. Por cada carta se definen sus atributos básicos y sus 3 listas de efectos, con los correspondientes atributos de cada uno. El sistema está diseñado para que únicamente acceda a una biblioteca de cartas, definida en este archivo “Cartas.XML”.
- Plantillas: Son ficheros independientes que definen un modelo de partida, mediante referencias a cartas del repositorio, que formaran el conjunto de pilas de suministro que serán accesibles en una partida, Pueden existir varios ficheros ya que cada uno responde a un modelo de partida diferente. La multiplicidad con respecto al fichero Cartas es 1 N, siendo n los ficheros plantilla y uno el repositorio de cartas. A pesar de poder existir infinitas plantillas, solo se puede usar una por partida.

Diseño SGBD

XML: Apostando por la claridad decidimos crear dos clases para abarcar completamente la gestión de la base de datos: una que se encargará de la lectura y otra de la escritura. En este apartado definiremos las funcionalidades que ofrecen cada una y las decisiones de diseño que se tomaron a la hora de implementarlas.

Lector:

- **Funcionalidad principal:** Obtener de la base de datos la información que necesita el sistema para su correcto funcionamiento.

- **Funcionalidades:**
  - Cargar plantilla: Para cada partida se necesitan unas pilas de suministro. Dichas pilas vienen codificadas en un archivo de plantilla XML. Esta función, tras recibir la ruta por parámetro se encarga de iniciar los procesos necesarios para la creación de dichas pilas.

  - Cargar cartas: Una vez extraídas qué pilas se dispondrán en la partida, es necesario cargar la plantilla de cartas para obtener la definición concreta de dichas pilas, ya que en plantilla solo se guarda una referencia a la carta y su número de ejemplares. Se extraen las cartas referenciadas en plantillas y se crea un objeto Pila a partir de cada TipoCarta, creado al leer los modelos del fichero cartas.XML. Las cartas están divididas por tipos, de Reino, de Territorio y de Tesoro, tanto en la plantilla como en el repositorio de cartas. Con esto no solo se consigue una mejor ordenación si no también un aumento del rendimiento en la carga de cartas. Esto es debido a que se minimizan las iteraciones fallidas en el proceso de *"matching"* entre la referencia de la plantilla y las cartas del repositorio.

- **Decisiones de diseño:** La decisión más relevante aquí es quizás el haber separado la carga de plantillas de la carga de cartas dejando abierta la posibilidad en un futuro de que la carga de cartas no esté directamente ligada a una plantilla de inicio y pueda usarse para un salvado/carga de partidas anteriores.

Escritor

- **Funcionalidad principal:** Escribir en la base de datos la información que el usuario o el sistema quieran crear modificar para eventos o procesos posteriores.

- **Funcionalidades:**
  - Cargar Cartas: A diferencia de la clase lector, este método no sirve para cotejar referencias y crear pilas si no para cargar todas las cartas de los tres diferentes tipos para poder añadirlas, borrarlas o editarlas de una plantilla creada.

  - Cargar plantilla: Este método carga la plantilla para su edición ya sea de nueva creación o una seleccionada creada previamente. Este método, a la vez que carga una plantilla cargará a la vez las cartas disponibles a las que podrán hacer referencia las pilas y llamará al menú de edición de pilas implementado en esta misma clase.

  - Menús de escritura: Podemos incluir dos menús en este apartado: el de la gestión de pilas y el de edición. El objetivo es el mismo y es ofrecer todas las opciones posibles al usuario: impresión por consola de cartas disponibles, pilas creadas, descripción de cartas, etc. Este menú que se apoya directamente en la clase IOMenus.

  - Edición de pilas: A través de un menú tendremos acceso completo a la plantilla que tenemos cargada donde podremos visualizar las cartas disponibles por tipos a la vez que las pilas existentes en la plantilla también organizadas por tipos. Podremos añadir cualquier pila de cualquier carta existente en la base de datos así como borrar o modificar pilas ya existentes.

- **Decisiones de diseño:** La decisión más importante es la creación de esta clase, en la que automatizamos la creación y edición de plantillas de juego. Si bien en un principio se pensó que el diseño era lo suficientemente sencillo como para hacerlo manualmente, creímos que el hecho de crear este módulo le daba un plus al proyecto que podría ser de gran utilidad para el usuario medio.

3.5 Implementación

En esta última parte del aparatado de desarrollo analizaremos más profundamente y desde el punto de vista de un desarrollador.

3.5.1 Base de datos

En este apartado analizaremos la implementación de los dos ficheros principales que componen la base de datos de nuestro programa. Y que ya estudiamos en el punto de la memoria dedicado al diseño del proyecto...

3.5.1.1 Cartas

Este fichero se encarga de almacenar todas las cartas a las que tendremos acceso a la hora de crear diferentes plantillas. Las cartas están organizadas por tipos: reino (acción), tesoro y territorio.

- **Etiquetas:**
  - Carta: Define los atributos principales de la carta, que se introducen por parámetro y que sirven de referencia para el fichero plantillas.
    - Nombre: El nombre de la carta que se traduce en un String y que sirve de referencia para unir pilas y cartas.
    - Tipo: El tipo de la carta “territorio”, “acción”, “tesoro”, “ataque” y “reacción”.
    - Coste: El coste en tesoros de la carta.
    - Gestión: Un 0 o un 1 dependiendo de si la carta se descarta o elimina al usarse.
    - Descripción: La descripción de la carta.
  - Efecto: Define los atributos principales del efecto. Que se usan para identificar el objeto correcto a crear cuando el programa lee el fichero y como parámetros.
    - Id: Identificador del efecto, se usa para identificar el objeto a crear.
    - Cantidad: El parámetro entero que define valores cuantitativos en los efectos, por ejemplo en el efecto robar, cuántas cartas se roban.
    - Objetivos: Define si el efecto ha de aplicarse en el poseedor de la carta, en el resto de jugadores o en todos los jugadores incluido el poseedor.
    - String: Este atributo no está presente en todos los efectos que se definen en el XML. En algunos efectos se necesita un atributo más para definir tipos, atributos numéricos extra o nombres.
  - Efectos territorio: Lista de efectos que se leerán si la carta se juega en la fase de territorios.
  - Efectos tesoro: Lista de efectos que se leerán si la carta se juega en la fase de tesoro.
  - Efectos acción: Lista de efectos que se leerán si la carta se juega en la fase de acción.

- **Decisiones de diseño:** Se decidió crear una clase para las Cartas externa a las plantillas con el objetivo de separar con mucha más claridad el contenido que necesita el programa, facilitando el entendimiento del programa por parte del usuario y facilitando el trabajo de desarrollo. La multiplicidad de este fichero es 1. El sistema solo se basa en un único repositorio de cartas, que es el definido por el fichero Cartas.XML

3.5.1.2 Plantillas

Este fichero se encarga de almacenar una plantilla de juego para una partida de Dominion, guardando referencias a las cartas almacenadas en el fichero cartas e indicando la cantidad de cartas que habrá disponibles en la partida.

- **Etiquetas:**
  - Tablero: Define los atributos principales de la carta, que se introducen por parámetro y que sirven de referencia para el fichero plantillas.
    - Reino, Tesoro, Territorios: Específica el tipo de carta, y en qué grupo habrá que buscar la referencia en el fichero cartas.
  - Pila: Define los atributos principales del suministro de cartas del tablero.
    - Id: Identificador del efecto, se usa para identificar el objeto a crear en la lista de cartas.
    - Cantidad: El parámetro entero que define la cantidad de cartas disponibles en el suministro.
- **Decisiones de diseño:** Es un diseño sencillo y eficiente. Si bien en un principio se planteó hacer cantidades de cartas variables según el número de jugadores, al final se optó por dar la mayor versatilidad posible usando el número de cartas deseado independientemente de los jugadores.

3.5.2 Gestor de Plantillas

En este apartado vamos a analizar los puntos clave de la implementación de las clases que gestionan la base de datos.

3.5.2.1 Lector

Es la clase del paquete que desarrolla la función de acceder a los archivos XML de la base de datos del programa.

- Constructor: Tiene un parámetro *boolean* que indica si se quiere que se impriman las trazas de lectura.
- Atributos: La lista de pilas del tablero que se crearán al hacer la carga, el tablero de la partida y el *boolean* que indica si se imprimen las trazas o no.
- Métodos públicos: Los referentes a la carga de plantilla y cartas.
- Métodos privados: El método que crea específicamente cada pila y el de impresión.

3.5.2.2 Escritor

Es la clase del paquete que desarrolla la función de crear y editar los archivos XML de la base de datos del programa. Específicamente crea y edita Plantillas.

- Constructor: El constructor por defecto que inicializa todos los objetos de los atributos.
- Atributos: Un *boolean* que indica si imprimir las trazas o no, así como 3 listas para las cartas de cada tipo y otras 3 para las listas de pilas que se carguen desde la plantilla.
- Métodos públicos: Para cargar en buffer una plantilla y para cargar el menú de gestión de pilas.
- Métodos privados: Los métodos de impresión y las funciones de edición de pilas (añadir, editar, borrar).

3.5.3 Motor de Juego

En este apartado analizaremos la implementación de las clases más relevantes del motor de juego.

3.5.3.1 Juego

Es la clase del paquete que desarrolla la función de gestionar las partidas y ser la puerta de entrada para la gestión de la base de datos.

- Constructor: El constructor por defecto
- Atributos: De tipo que String que guarda la ruta de la plantilla que se usará en la partida y un *boolean* que indica si se ha lanzado una partida modo test.
- Métodos públicos: El main que carga el programa y en concreto el menú del juego.
- Métodos privados: Ninguno.

3.5.3.2 Partida

Es la clase del paquete que gestionará el inicio de una partida, los turnos y su finalización delegando la mecánica interna en el tablero, las cartas y los jugadores.

- Constructor: Posee dos constructores, en uno se pasan los jugadores la plantilla y el IOMenus por parámetro mientras que en el otro la plantilla no se pasa, con lo que carga una por defecto.
- Atributos: Una referencia a IOMenus, otra al objeto partida que se cree y uno de tipo que String que guarda la ruta de la plantilla que se usará en la partida.
- Métodos públicos: Los constructores
- Métodos privados: Inicializar la partida y el manejo de turnos, además del de finalización de partida, como método de seguridad se ponen todos los métodos privados para que la clase sea la única que puede gestionar las partidas a ese nivel.

3.5.3.3 Tablero

Esta clase es responsable de toda la mecánica interna al juego en lo relativo a gestión de cartas y efectos, teniendo que pasar todos los intercambios o procesos relacionados por esta clase.

- Constructor: Posee un constructor que recibe la lista de jugadores así como la lista de pilas de suministro disponibles.
- Atributos: El quizás más relevante es la referencia al jugador actual que se actualiza al inicio de cada turno, además de la fase en la que se encuentra la partida. Tiene una lista en la que se guardan las pilas de suministros, un *boolean* que indica si la partida aún está en funcionamiento y una lista de cartas eliminadas.
- Métodos públicos: Prácticamente todos, ya que es la clase que más llamadas tiene por parte de jugadores, inteligencias y cartas.
- Métodos privados: El método de aplicar efecto. Como protegidos se han definido los métodos que cambian la fase del juego, para que únicamente clases del paquete puedan hacerlo.

3.5.3.4 Jugador

Es la clase que representa al jugador dentro de la partida, sea humano o no, ya que de cara al motor es lo mismo. La diferencia radica en la inteligencia que lo controla, establecida por parámetro al crear el jugador.

- Constructor: Un constructor que recibe el nombre del mismo, el IOMenus y la inteligencia que controlará el jugador así como inicializa todas las variables y establece el jugador en la inteligencia que lo controla.
- Atributos: Las listas de cartas en posesión del jugador, su nombre y la referencia a la inteligencia que lo controla.
- Métodos públicos: Prácticamente todos, incluidos la aplicación de efectos, ya que la mayoría se invocan desde otras clases, como la inteligencia, el tablero o las cartas.
- Métodos privados: Los métodos de activación de fases del turno y el de jugar carta.

3.5.3.5 TipoCarta

Es la que representa la abstracción de la carta, alberga una lista de efectos y los atributos que definen a la carta.

- Constructor: Tiene dos constructores, unos que recibe los valores por parámetro y otro que recibe un elemento XML y que el constructor decodifica para crear el objeto.
- Atributos: Todos los que definen una carta, su coste su nombre su tipo sus listas de efectos, si es prioritario.
- Métodos públicos: Prácticamente todos, ya que los TipoCarta son objetos que se acceden usualmente desde la carta que a su vez es accedida por el esto de clases.
- Métodos privados: El método que gestiona las listas de efectos y que invoca a Tipos para su creación.

3.5.4 InterfazJugador

Este paquete que se basa principalmente en dos clases, o dos modelos de clases. Una la clase Inteligencia y sus clases que heredan, que define las acciones específicas a la que deberán responder durante el juego. La otra, IOMenus es la que gestiona toda la interacción Humano Ordenador cuando la inteligencia es de tipo Humano.

3.5.4.1 Inteligencia

Es una clase abstracta que define el esqueleto de funciones a implementar para poder cumplir los requisitos o peticiones del motor de juego de manera satisfactoria.

- Constructor: El constructor por defecto en ambas clases que heredan.
- Atributos: La referencia al tablero de la partida y al jugador que controlen.
- Métodos públicos: Todos están relacionados con la toma de decisiones. Se diferencian por los parámetros que reciben, por los valores que devuelven y su dato más relevante es quizás la implementación de switch internos para especificar más las respuestas.
- Métodos privados: Ninguno.

3.5.4.2 IOMenus

Es la clase que gestiona todas las entradas del sistema, tanto para la introducción de opciones en los menús como en la introducción de comandos a la hora de tomar decisiones durante el turno del jugador controlado por un humano.

- Constructor: El constructor por defecto, inicializa el objeto de Scanner.
- Atributos: El tablero de la partida y el objeto *Scan* que permite la introducción de comandos por consola.
- Métodos públicos: Todos menos los que activa el menú de escape.
- Métodos privados: Los que activa el menú de escape, que son métodos auxiliares para conocer el estado de la partida, el jugador actual y las pilas disponibles.

3.5.5 Tecnologías

Este apartado explica el software que se ha usado para elaborar el proyecto, haciendo mención a sus características, puntos fuertes y a las razones que nos llevaron a usarlos en dicho desarrollo.

3.5.5.1 Eclipse IDE

Eclipse es un IDE (entorno de desarrollo integrado) creado bajo código libre que se ideó con el objetivo de crear una plataforma robusta con todas las características y calidad requeridas para poder desarrollar herramientas altamente integradas.

Entre sus características está la alta modularidad, que aunque en parte inherente a su naturaleza de código libre, es uno de los aspectos más significativos de la plataforma. Tanto consorcios de empresas como grandes grupos de desarrollo y programadores autónomos además de otros tantos grupos de interés, han colaborado y colaboran en el continuo desarrollo de una plataforma que da cabida a varios lenguajes, herramientas de Project Management, subversión, módulos de acoplamiento; en definitiva a todo lo que pudiésemos necesitar para desarrollar un producto software de calidad industrial.

Nosotros decidimos usar esta herramienta por la familiaridad adquirida durante los años de carrera desarrollando en lenguaje Java, sus grandes posibilidades de ampliación y extensión a módulos como JUnit, Subversión o UML Design.

3.5.5.2 JDom2.0

JDom es una API desarrollada para dar tratamiento a archivos XML, y es específica para Java. Es un modelo similar al que planteó W3C con DOM, con la salvedad de que JDom es específico para java y DOM se creó para que fuese independiente del lenguaje.

El funcionamiento de ambos es similar creando un árbol de nodos con los elementos del archivo XML tras realizar el parseo del mismo. Esto es interesante y es una de las razones principales por las que se escogió JDom frente a otras APIs de gestión de ficheros XML, y es que, con JDOM cargamos todos los elementos del fichero en el buffer, en vez de realizar lecturas secuenciales.

Con esto se consigue hacer una única lectura y pudiendo gestionar los elementos desde el programa sin tener que realizar más accesos al fichero. Con esta decisión ganamos en varios aspectos, primero en rapidez al no tener que acceder más veces al fichero para leer o clasificar la información y dos en accesibilidad, las mejoras en accesibilidad y organización se refieren al acceso al árbol completo de una sola vez; poder definir con más claridad los elementos en la implementación y no tener que llamar a funciones auxiliares cada vez que haya que extraer nueva información y posteriormente organizarla.

4 Resultados

Como parte final del análisis del proyecto, en este capítulo nos centraremos en los resultados obtenidos en puntos muy concretos, como son la flexibilidad y la usabilidad. Además de testear la partida de Dominion en sí, probando que todos y cada uno de los efectos implementados actúe como debería, así como que la partida sigue el orden correcto de turnos, robo y finalización de partida.

4.1 Flexibilidad

En cuestión de flexibilidad los resultados los analizaremos desde dos perspectivas: las de flexibilidad como plataforma y la flexibilidad para implementar inteligencias que controlen un jugador.

4.1.1 Flexibilidad de ampliación

Uno de los objetivos marcados para el proyecto fue conseguir una plataforma fácilmente extensible, no solo para ampliaciones del mismo Dominion debido a sus expansiones comerciales, si no también que la plataforma fuera apta para albergar otro juego basado en cartas sin demasiados cambios.

Cartas

En proyectos similares las cartas eran entes complejos programados como objetos únicos, mientras que en el nuestro las cartas son portadoras de efectos. Estos efectos son unidades más pequeñas, que pueden actuar de independientemente o agrupados en una lista.

Cada carta de las que existen en la entrega básica de Dominion pueden ser creadas agrupando los efectos implementados en nuestro programa. Con esto, aseguremos que no solo nuestro proyecto es fidedigno a Dominion si no que, además será fácilmente ampliable mediante más agrupaciones de efectos o mediante la adición de algún efecto nuevo para cubrir alguna nueva funcionalidad de nueva implementación. Las cartas se crean de manera mucho más sencilla que en otros proyectos ya que se definen un archivo XML muy claro y explícito.

Mecánica

Además sabiendo que los que realmente definen el juego son los usos de los efectos, podríamos adaptar cualquier juego de cartas a nuestra plataforma. Si bien tendría algo más de complejidad que ampliar Dominion (fases del turno, número de acciones/compras, etc) no llevaría tiempo en exceso adaptar la plataforma, con efectos nuevos que fueran capaces de cubrir el otro juego, las fases del turno, capacidades de compra y juego de acciones, despacho de cartas post-uso, gestión del mazo.

- Las fases están implementadas de manera independiente, con lo cual podrían añadirse quitarse o cambiarse como se necesitase sin necesariamente afectar a las otras.
- El número de acciones/compras se restablece en cada fin de turno, con lo cual tampoco tendría mucha problemática.
- Lo mismo con la gestión del mazo, cuándo y cómo se barajan las cartas, cuántas cartas se roban, y similares características serían de fácil edición.
- La gestión de cartas es una función independiente que tras la aplicación de efectos decide cuál es el destino de la carta, dependiendo del juego tendría un destino u otro.

4.1.2 Flexibilidad en el diseño de IAs

La flexibilidad reside en que el funcionamiento de la plataforma está definido en el motor, de manera totalmente independiente de la toma de decisiones por parte de la inteligencia. Dicho funcionamiento es una caja opaca que solo pide respuestas, es decir, mientras se gestionan estas respuestas el cómo se hagan es indiferente. Nuestro diseño comparado con otros sistemas es totalmente innovador. En los modelos actuales la toma de decisiones estaba totalmente incrustada en el motor del juego, o contaba con un sistema de prioridades muy rígido ya implementado, o incluso se regía por un sistema de reglas condicionantes que a la larga también carecían de flexibilidad.

Con nuestro diseño, simplemente creando una clase que herede de inteligencia se pueden implementar métodos que gestionen la información a gusto del desarrollador, obteniendo información del tablero, del jugador, guardar cartas jugadas, compradas, etc.…

Con este sistema no solo hemos conseguido independizar la IA del sistema sino además ofrecer total libertad de creación a usuarios y desarrolladores.

4.2 Usabilidad

Considerando usabilidad como la facilidad con que las personas pueden utilizar una herramienta particular o cualquier otro objeto fabricado por humanos con el fin de alcanzar un objetivo concreto, creemos que la usabilidad es otro de los puntos fuertes del sistema, no solo en el momento de jugar sino también a la hora de la edición de cartas y plantillas.

4.2.1 Usabilidad en el juego

Si hablamos del juego como la parte que solo se refiere a la partida en sí creemos que el grado de usabilidad es bastante alto incluso contando con una interfaz de manejo por consola.

- El sistema comprueba las acciones/compras/tesoros restantes y en caso de respuesta negativa el sistema salta directamente a la fase siguiente sin necesidad de interactuar.
- Se puede saltar de fase de turno con solo una introducción de comando.
- Se pueden usar todos los tesoros con un solo comando.
- Se puede seleccionar la carta a comprar (y comprarla) con solo un comando.
- En cualquier momento del turno el jugador puede acceder a la información relevante de la partida, su estado y a la descripción de las pilas de suministros.
- Siempre que se pueda elegir entre una lista de cartas o decisiones del estilo de no elegir ninguna o elegir todas se podrá elegir cualquiera de las opciones con solo un comando.
- Se ha reducido al mínimo las salidas por consola, dejando únicamente la información útil con el fin de optimizar la lectura y facilitar el seguimiento de la partida.

4.2.2 Usabilidad en los menús

En este apartado analizaremos la usabilidad de los menús de creación de partidas y de gestión de bases de datos.

- Al arrancar el programa aparece directamente el menú de partidas.
- El menú de partidas está diseñado para que con la introducción dé como máximo 2 comandos puedas acceder/tomar cualquier opción/decisión.
- Con dos comandos puedes crear un jugador y elegir el tipo (humano o controlado por IA)
- Con dos comandos puedes listar todas las plantillas disponibles y seleccionar una.
- Para acceder al menú de pilas accederemos directamente con 2/3 comandos con una plantilla cargada en el buffer.
- La gestión de pilas se organiza por tipos para que la consola se recargue menos con opciones de cartas y a la vez ser más intuitivo.
- Antes de tomar decisiones de edición de pilas se puede ver las cartas disponibles en las base de datos y en la plantilla de cartas.

4.3 Pruebas

En este apartado comprobaremos que el funcionamiento del juego Dominion es el correcto, tanto en lo referente al motor o mecánica de la partida y todo lo que abarca (fases del turno, uso y compra de cartas, gestión de turnos y finalización de la partida), así como las repercusiones de los efectos en la partida, comprobando que su implementación cumple de manera fidedigna los efectos reales del juego original.

4.3.1 Mecánica de la partida

Analizaremos cada caso específico y explicaremos porque concluimos que el resultado es correcto.

Turnos

Se ha comprobado que siempre se sigue el orden correcto en la lista de jugadores y que cada uno sigue la secuencia correcta de fases en el turno. Para comprobar esto se modificó el transcurso entre turnos para que el usuario tuviera que pulsar una tecla cada vez que el turno volvía al primer jugador. Gracias a esto se pudo comprobar no solo que las fases seguían el orden apropiado además de cumplir las condiciones para uso de acciones, tesoros y adquisición de cartas si no que los turnos se asignaban a los jugadores en el orden apropiado.

Reacción

Cuando un jugador ejecuta una acción de ataque, el proceso de ejecución es especial, ya que el efecto se ejecutará únicamente en los jugadores que no jueguen una reacción. Que es una carta especial que evita que el jugador que la juega sea atacado.

Reacción	
Descripción	El jugador 1 juega una carta de ataque por la cual el jugador [0] deberá descartarse hasta quedarse con 3 cartas en mano, el sistema detecta que tiene reacciones y le pregunta si quiere jugarlas. Y jugándola evita el ataque.
Estado previo	`El jugador jugador [1] juega la carta Milicia.`

Estado posterior	`¿Quieres jugar una reacción? 1: Si 0: No`
	`1`
	`0) Foso`
	`Elige una reaccion`
	`o pulsa h para funciones especiales`
	`0`
	`El jugador jugador [0] juega la carta Foso`

Tabla 48. Reacción

Finalización de la partida

Para comprobar si la partida finalizaba correctamente tanto el momento de finalizar como si se contabilizaban correctamente los puntos, podemos leer por consola las pilas que se han agotado así como la contabilización de puntos de victoria de cada jugador. Para comprobarlo paramos el juego cada vez que una partida finaliza y muestra por pantalla la resolución de puntos.

Finalización	
Descripción	Se comprueba las pilas que están vacías para ver cuando finaliza la partida.
Finalización normal	La pila Capilla se ha acabado La pila Sotano se ha acabado La pila Foso se ha acabado El jugador ddd juega la carta Finca Se aplica el efecto --------> .EfectosTerritorio parametro: 1 … El jugador ddd juega la carta Maldicion Se aplica el efecto --------> .EfectosTerritorio parametro: -1 El jugador aaa juega la carta Finca Se aplica el efecto --------> .EfectosTerritorio parametro: 1 … El jugador bbb juega la carta Finca Se aplica el efecto --------> .EfectosTerritorio parametro: 1 … El jugador bbb juega la carta Maldicion Se aplica el efecto --------> .EfectosTerritorio parametro: -1 El jugador bbb juega la carta Maldicion Se aplica el efecto --------> .EfectosTerritorio parametro: -1 El jugador bbb juega la carta Finca Se aplica el efecto --------> .EfectosTerritorio parametro: 1 0) aaa Territorios:4 Turnos: 38 1) bbb Territorios:3 Turnos: 38 2) ddd Territorios:2 Turnos: 38 Se acabo
Finalización por prioridad (Se usaron cierto número de pilas como prioritarias para hacer la prueba)	La partida finaliza porque la pila Foso se ha acabado. El jugador dee juega la carta Maldicion Se aplica el efecto --------> .EfectosTerritorio parametro: -1 El jugador dff juega la carta Finca Se aplica el efecto --------> .EfectosTerritorio parametro: 1 0) dff Territorios:1 Turnos: 30 1) dee Territorios:-1 Turnos: 31 Se acabó.

Tabla 49. Finalización

4.3.2 Efectos

Para realizar este test se ha implementado un modo de prueba llamado modo test. En este modo test podemos participar como jugador y tomar las decisiones con respecto a que cartas jugar. En este modo de test se proveerá a los jugadores del máximo de cartas posibles por turno, así como un número ilimitado de acciones para poder jugar todas las cartas de la mano.

En este método que dispondrá de un modo de impresión de trazas más completo podremos comprobar si el funcionamiento de los efectos es el esperado, comprobando los estados de cada jugador antes y después de la aplicación del efecto, así como también nos facilitará la depuración de código dando mayor rapidez y accesibilidad al código que si tuviéramos que montar una partida cada vez eligiendo jugadores, plantilla, comprando cartas, etc.…

Añadir que el juego consume la acción antes que aplicar los efectos, ya que la acción viene relacionada con jugar la carta no con aplicar sus efectos.

Identificador	Efecto Acción
Descripción	El jugador sobre el que se aplica el efecto puede jugar un número específico de acciones más.
Parámetros	Cantidad indica el número de acciones a sumar.
Estado previo jugador	Estado del jugador jugador [0] antes de jugar la carta JUGADOR [0] tiene: Tesoros: 0 Acciones: 24 Compras: 1
Estado posterior jugador	Se aplica el efecto --------> .EfectosAccion `parametro: 2` Estado del jugador jugador [0] despues de jugar la carta JUGADOR [0] tiene: Tesoros: 0 Acciones: 26 Compras: 1

Tabla 50. Efecto Acción

Identificador	Efecto Compra
Descripción	El jugador sobre el que se aplica el efecto puede ejercer un número específico de compras más.
Parámetros	Cantidad indica el número de compras a sumar.
Estado previo jugador	Estado del jugador jugador [0] antes de jugar la carta JUGADOR [0] tiene: Tesoros: 1 Acciones: 26 Compras: 1
Estado posterior jugador	Se aplica el efecto --------> .EfectosCompra Estado del jugador jugador [0] despues de jugar la carta JUGADOR [0] tiene: Tesoros: 1 Acciones: 26 Compras: 2

Tabla 51. Efecto Compra

Identificador	Efecto Tesoro
Descripción	El jugador sobre el que se aplica el efecto obtiene tesoros extra para usar en la fase de compra.
Parámetros	Cantidad indica el número de tesoros a sumar.
Estado previo jugador	Estado del jugador jugador [0] antes de jugar la carta JUGADOR [0] tiene: Tesoros: 0 Acciones: 25 Compras: 1
Estado posterior jugador	Se aplica el efecto --------> .EfectosTesoro `parametro: 2` Estado del jugador jugador [0] despues de jugar la carta JUGADOR [0] tiene: Tesoros: 2Acciones: 25 Compras: 1

Tabla 52. Efecto Compra

Identificador	Efecto Aventurero
Descripción	El jugador sobre el que se aplica el efecto va robando cartas descartando las que no sean tesoros y añadiendo a su mano las que si lo sean.
Parámetros	Cantidad indica el número de tesoros máximos a añadir a la mano.
Estado previo jugador	
	Mano del jugador [0] 0:Ducado 1:Burocrata 2:Taller 3:Cobre 4:Finca 5:Milicia 6:Cobre 7:Prestamista 8:Cobre 9:Remodelar 10: Plata
Estado posterior jugador	
Se aplica el efecto --------> .EfectosAventurero Se descarta la carta Finca por el efecto Aventurero Se descarta la carta Sotano por el efecto Aventurero Se descarta la carta Biblioteca por el efecto Aventurero Se descarta la carta Maldicion por el efecto Aventurero Se descarta la carta Salon del Trono por el efecto Aventurero	Estado del jugador jugador [0] despues de jugar la carta 0:Ducado 1:Burocrata 2:Taller 3:Cobre 4:Finca 5:Milicia 6:Cobre 7:Prestamista 8:Cobre 9:Remodelar 10:Plata 11: Cobre Tipo: Tesoro Coste: 0 12: Cobre Tipo: Tesoro Coste: 0

Tabla 53. Efecto Aventurero

Identificador	Efecto Biblioteca
Descripción	El jugador que juega la carta robará de su mazo cartas hasta tener X cartas en mano. Si la carta que roba es una de acción podrá descartarla y seguir robando.
Parámetros	Índica el número máximo de cartas que el jugador tendrá en mano.

Estado previo jugador

```
Estado del jugador jugador [2] antes de jugar
la carta                                        Mano del jugador [2]: Vacia
JUGADOR [2] tiene:   Tesoros: 2 Acciones: 27
Compras: 2
```

Estado posterior jugador

```
Se aplica el efecto -------->
.EfectosBiblioteca parametro: 7
Roba hasta que tengas 7 cartas en la mano,
descarta las acciones que no quieras
Maldicion
Robada Maldicion
Provincia
Robada Provincia
Leñadores
Plata
Robada Plata
Maldicion
Robada Maldicion
Cobre
Robada Cobre
Maldicion
Robada Maldicion
Plata
Robada Plata
Maldicion
Robada Maldicion
```

```
Mano del jugador [2]
0 Maldicion
1 Provincia
2:Plata
3 Maldicion
4 Cobre
5:Maldicion
6: Plata
7: Maldicion
```

Tabla 54. Efecto Biblioteca

Identificador	Efecto GanarCartaCosteFijo
Descripción	El jugador sobre el que se aplica el efecto obtiene una carta de un coste máximo indicado por parámetro.
Parámetros	Cantidad indica el coste máximo de la carta a obtener de los suministros.
Estado previo jugador	
El jugador jugador [0] juega la carta Banquete Estado del jugador jugador [0] antes de jugar la carta JUGADOR [0] tiene: Tesoros: 0 Acciones: 25 Compras: 1	Mano del jugador [0] 0: Prestamista 1: Foso 2: Cobre 3: Sala del Consejo 4: Espia 5: Ducado 6: Laboratorio 7: Cobre 8: Plata 9: Cobre 10: Bruja 11: Cobre
Estado posterior jugador	
Se aplica el efecto --------> .EfectosGanarCartaCosteFijo parametro: 5 Selecciona una carta de coste 5 o menos Se quieren obtener pilas del Tipo: Nombre: Coste: 5	… 19: Biblioteca Coste: 5 20: Remodelar Coste: 4 21: Canciller Coste: 3 22: Ducado Coste: 5 23: Finca Coste: 2 24: Maldicion Coste: 0 25: Cobre Coste: 0 26: Plata Coste: 3 Elige una pila a obtener: 26 **Creando carta: Plata**

Tabla 55. Efecto GanarCartaCosteFijo

Identificador	Efecto GanarCartaNombre
Descripción	El jugador sobre el que se aplica el efecto obtiene una carta con el nombre indicado por parámetro.
Parámetros	Cantidad indica el número de cartas que se obtienen.
Estado previo jugador	
Estado del jugador jugador [1] antes de jugar la carta JUGADOR [1] tiene: Tesoros: 0 Acciones: 0 Compras: 0	Mano jugador 1 … 8: Salon del Trono 9: Prestamista 10: Cobre 11: Capilla 12: Banquete 13: Cobre
Estado posterior jugador	
Se aplica el efecto --------> .EfectosGanarCartaNombre parámetro: 1 parámetro: Maldicion Selecciona una de carta de Maldición Se quieren obtener pilas del Tipo: Nombre: Maldicion Coste: 0 Creando carta: Maldición	

Tabla 56. Efecto GanarCartaNombre

Identificador	Efecto GanarCartaTipo
Descripción	El jugador sobre el que se aplica el efecto obtiene una carta con el tipo indicado por parámetro.
Parámetros	• Cantidad indica el número de cartas que se obtienen. • Indica el tipo de la carta a obtener.
Estado previo jugador	
Estado del jugador jugador [0] antes de jugar la carta JUGADOR [0] tiene: Tesoros: 3 Acciones: 27 Compras: 2	Mano del jugador [0] … 4: Finca 5: Cobre 6: Cobre 7: Mina
Estado posterior jugador	
Se aplica el efecto --------> .EfectosGanarCartaTipo parametro: 1 parametro: Accion Selecciona una de carta de Accion Se quieren obtener pilas del Tipo: Accion 0: Banquete Coste: 4 1: Festival Coste: 5 2: Aventurero Coste: 6 3: Capilla Coste: 2 4: Aldea Coste: 3 5: Leñadores Coste: 3 6: Laboratorio Coste: 5 7: Sala del Consejo Coste: 5 8: Taller Coste: 3 ..	Elige una pila a obtener: 8 Creando carta: Taller

Tabla 57. Efecto GanarCartaNombre

<table>
<tr><th>Identificador</th><th>Efecto Burócrata</th></tr>
<tr><td>Descripción</td><td>• El jugador que juega el efecto obtiene tantos tesoros de orden 2 como se indicara por parámetro, dichos tesoros se colocaran encima de su mazo.
• El resto de jugadores deberá descartarse de una carta de territorio cada uno, dicha carta irá a la cima de sus propios mazos.</td></tr>
<tr><td>Parámetros</td><td>Cantidad indica el número de cartas que se obtienen.</td></tr>
<tr><td colspan="2">Estado previo jugador</td></tr>
<tr><td colspan="2">Estado del jugador jugador [0] antes de jugar la carta
JUGADOR [0] tiene: Tesoros: 4 Acciones: 27 Compras: 3

Mano del jugador 0:
…
7: Milicia
8: Ducado
9: Oro</td></tr>
<tr><td colspan="2">Estado posterior jugador</td></tr>
<tr><td colspan="2">Se aplica el efecto -------->
.EfectosBurocrata parametro: 1
Se quieren obtener pilas del Tipo: TesoroNombre: Plata Coste: 0
0: Plata Coste: 3
Elige una pila a obtener:
0
Creando carta: Plata
0: Plata Coste: 3
Elige una pila a obtener:
0
Creando carta: Plata

Elige una carta de territorio
El jugador jugador [1] se descarta de la carta Ducado
Elige una carta de territorio
El jugador jugador [2] se descarta de la carta Finca
Elige una carta de territorio
El jugador jugador [3] se descarta de la carta Ducado</td></tr>
</table>

Tabla 58. Efecto Burócrata

Identificador	Efecto DescartarCosteConcreto
Descripción	El jugador se tendrá que descartar de una carta que cumpla el requisito de coste en caso de que la tenga.
Parámetros	Se indica por parámetro las condiciones del descarte de tipo, nombre o coste.
Estado previo jugador	
Estado del jugador jugador [0] antes de jugar la carta JUGADOR [0] tiene: Tesoros: 0 Acciones: 25 Compras: 1	Mano del jugador [0] 0:Sotano Tipo: Accion Coste: 2 1:Aldea Tipo: Accion Coste: 3 2:Finca Tipo: Territorio Coste: 2 3:Cobre Tipo: Tesoro Coste: 0 4:MaldicionTipo:Territorio Coste:0 5:Banquete Tipo: Accion Coste: 4 6:Burocrata Tipo: Ataque Coste: 4 …
Estado posterior jugador	
Se aplica el efecto --------> .EfectosDescartarCosteConcreto parametro: 3 descartarte de una carta con un coste minimo de 3 Elige una carta a descartar: 0) Aldea 1) Banquete 2) Burocrata 3) Salon del Trono 4) Aventurero o pulsa h para funciones especiales	

Tabla 59. Efecto Descartar(Coste, nombre, tipo)

Identificador	Efecto DescartarNombre
Descripción	El jugador se tendrá que descartar de una carta que cumpla el requisito de nombre en caso de que la tenga.
Parámetros	Se indica por parámetro las condiciones del descarte nombre.
Estado previo jugador	
Mano del jugador [0] 0: Prestamista Tipo: Accion Coste: 4 1: Capilla Tipo: Accion Coste: 2 2: Burocrata Tipo: Ataque Coste: 4	
Estado posterior jugador	
Se aplica el efecto --------> .EfectosDescartarNombreConcreto parametro: 3 parametro: Cobre El jugador no tiene cartas de nombre Cobre Estado del jugador jugador [0] despues de jugar la carta	----------MANO DEL Mano del jugador [0] 0: Prestamista 1: Capilla 2: Burocrata

Tabla 60. Efecto Descartar(Coste, nombre, tipo)

Identificador	Efecto DescartarTipo
Descripción	El jugador se tendrá que descartar de una carta que cumpla el requisito de nombre, tipo o coste.
Parámetros	Se indica por parámetro las condiciones del descarte tipo.
Estado previo jugador	
Mano del jugador [0] 0: Prestamista Tipo: Accion Coste: 4 1: Capilla Tipo: Accion Coste: 2 2: Burocrata Tipo: Ataque Coste: 4	
Estado posterior jugador	
Se aplica el efecto -------> .EfectosDescartarTipoConcreto parametro: 3 parametro El jugador no tiene cartas de tipo Territorio Estado del jugador jugador [0] despues de jugar la carta	----------MANO DEL Mano del jugador [0] 0: Prestamista 1: Capilla 2: Burocrata

Tabla 61. Efecto Descartar(Coste, nombre, tipo)

Identificador	Efecto Remodelar
Descripción	El jugador que recibe el efecto podrá eliminar X cartas de su mano y obtener una carta con coste máximo de 2 unidades más que la carta eliminada. X se indica por parámetro.
Parámetros	Cantidad el número de cartas que se pueden eliminar/ obtener.
Estado previo jugador	
JUGADOR [0] tiene: Tesoros: 0 Acciones: 25 Compras: 1 Mano del jugador 0	… 5: Finca 6: Cobre 7: Cobre 8: Sala del Consejo 9: Laboratorio 10: Biblioteca 11: Cobre
Estado posterior jugador	
Elige una carta para eliminar y obtener una carta de coste 2 unidades mayor a esta o pulsa h para funciones especiales 4 Se quieren obtener pilas del Tipo: Nombre: Coste: 5 … 19: Biblioteca Coste: 5 20: Remodelar Coste: 4 21: Canciller Coste: 3 22: Ducado Coste: 5 23: Finca Coste: 2 24: Maldicion Coste: 0 25: Cobre Coste: 0 26: Plata Coste: 3	Elige una pila a obtener: 16 Creando carta: Prestamista Estado del jugador jugador [0] despues de jugar la carta JUGADOR [0] tiene: Tesoros: 0 Acciones: 25 Compras: 1

Tabla 62. Efecto Remodelar

Identificador	Efecto Descarte
Descripción	El jugador tendrá que descartarse de tantas cartas como se indique por parámetro.
Parámetros	Se indica por parámetro el número de cartas que el jugador tiene que descartarse.
Estado previo jugador	
Estado del jugador jugador [0] antes de jugar la carta JUGADOR [0] tiene: Tesoros: 3 Acciones: 27 Compras: 2 Mano del jugador [0] … 6: Finca 6: Cobre 7: Cobre 8: Mina 9: Cobre	
Estado posterior jugador	
Se aplica el efecto --------> .EfectosDescarte parametro: 1 JUGADOR [0] tiene: Tesoros: 3 Acciones: 27 Compras: 2 Elige una carta a descartar … 7) Cobre 8) Mina 9) Cobre o pulsa h para funciones especiales 9	Mano del jugador [0] … 5: Finca 6: Cobre 7: Cobre 8: Mina

Tabla 63. Efecto Descarte

Identificador	Efecto DescarteMano
Descripción	El jugador tendrá que descartarse hasta quedarse con el número de cartas que se indica por parámetro.
Parámetros	Se indica por parámetro el número de cartas que el jugador debe tener en la mano tras el descarte.
Estado previo jugador	
Se aplica el efecto --------> .EfectosDescarteMano parametro: 3 Descartarte de cartas hasta quedarte con 3 cartas en la mano JUGADOR [3] tiene: Tesoros: 0 Acciones: 0 Compras: 0	Mano del jugador [3] 0: Laboratorio 1: Finca 2: Prestamista 3: Mercado 4: Sala del Consejo 5: Canciller 6: Finca …
Estado posterior jugador	
Mano del jugador [3] 0: Finca 1: Oro 2: Milicia	

Tabla 64. Efecto DescarteMano

Identificador	Efecto Descarte Mazo
Descripción	El jugador tendrá que descartar el mazo completo.
Parámetros	
Estado previo jugador	

`Estado del jugador jugador [0] antes de jugar la carta`
`JUGADOR [0] tiene:  Tesoros: 3 Acciones: 27 Compras: 2`
`Mano del jugador [0]`
`…`
`6:  Cobre`
`7:  Cobre`
`8:  Mina`
`9:  Cobre`	

Estado posterior jugador	

`Se aplica el efecto -------->`
`.EfectosDescartarMazo parametro: 2`
`Poniendo mazo en la pila de descartes`	

Tabla 65. Efecto Descartar mazo

Identificador	Efecto Descarte Mazo opcional
Descripción	El jugador podrá descartarse del mazo completo.
Parámetros	
Estado previo jugador	
`Estado del jugador jugador [0] despues de jugar la carta` `JUGADOR [0] tiene:  Tesoros: 11 Acciones: 29 Compras: 4`	
Estado posterior jugador	

`Se aplica el efecto -------->`
`.EfectosDescartarMazoOpcional parametro: 1`
`¿Quieres descartar el mazo? 1: Si 0: No`
`1`	

Tabla 66. Efecto Descarte mazo opcional

Identificador	Efecto Repetir acción
Descripción	El jugador elige una carta de acción de su mano que se jugará X veces.
Parámetros	Las veces que se repite la acción.
Estado previo jugador	`Estado del jugador jugador [0] antes de jugar la carta.` `JUGADOR[0] tiene:Tesoros: 0 Acciones:25 Compras: 1`

Estado posterior jugador	`Se aplica el efecto -------->.EfectosRepetirAccion parametro: 1`
	
	`0) Leñadores`
	`1) Aldea`
	`2) Festival`
	`3) Aventurero`

Tabla 67. Efecto Repetir acción

Identificador	Efecto Sótano
Descripción	El jugador podrá descartarse de tantas cartas como quiera y obtener el mismo número de cartas.
Parámetros	No tienen efecto.
Estado previo jugador	
El jugador jugador [0] juega la carta Sotano Estado del jugador jugador [0] antes de jugar la carta JUGADOR [0] tiene: Tesoros: 0 Acciones: 25 Compras: 1 Mano del jugador [0] 0: Maldicion 1: Mina 2: Cobre 3: Mercado	4: Leñadores 5: Festival …
Estado posterior jugador	
Se aplica el efecto --------> .EfectosSotano parametro: 1 Robaras tantas cartas como descartes hagas Elige las cartas que quieres descartar 0) Maldicion 1) Mina 2) Cobre 3) Mercado 4) Leñadores 5) Festival … 2	Elige las cartas que quieres descartar 0) Maldicion 1) Mina 2) Mercado 3) Leñadores 4) Festival … 2 Mano del jugador [0] 0: Maldicion 1: Mina 2: Leñadores 3: Festival 4: Foso 5: Finca

Tabla 68. Efecto Sótano

Identificador	Efecto Mina
Descripción	El jugador podrá descartarse de un tesoro y obtener otro de orden superior, del mismo o inferior.
Parámetros	
Estado previo jugador	
El jugador jugador [0] juega la carta Mina Estado del jugador jugador [0] antes de jugar la carta JUGADOR [0] tiene: Tesoros: 4 Acciones: 27 Compras: 3	Mano del jugador [0] 0: Plata 1: Finca 2: Cobre 3: Cobre …
Estado posterior jugador	
Elige un tesoro a eliminar 0) Plata 1) Cobre 2) Cobre 3) Cobre 4) Cobre o pulsa h para funciones especiales 0 Por el efecto Mina eliminamos Plata Selecionamos carta por el efecto mina Se quieren obtener pilas del Tipo: TesoroNombre: Coste: 6 0: Cobre Coste: 0 1: Plata Coste: 3 2: Oro Coste: 6	Elige una pila a obtener: 2 Creando carta: Oro Oro Ha sido adquirida por el efecto Mina Mano del jugador [0] … 8: Milicia 9: Ducado 10:Burocrata 11:Oro

Tabla 69. Efecto Mina

Identificador	Efecto Jardines
Descripción	Gana tantos puntos de victoria como cartas tenga en el mazo.
Parámetros	
Estado previo jugador	
El jugador sss juega la carta Jardines sss ->> Territorios: 15	
Estado posterior jugador	
Se aplica el efecto --------> .EfectosJardines parametro: 1 El jugador sss juega la carta Finca	sss ->> Territorios: 22

Tabla 70. Efecto Jardines

Identificador	Efecto Prestamista
Descripción	El jugador podrá eliminar una carta concreta y obtener un número de tesoros que se especifica por parámetro.
Parámetros	• El nombre de la carta que se debe eliminar. • Los tesoros que se obtienen al eliminarla.
Estado previo jugador	
Estado del jugador jugador [0] antes de jugar la carta JUGADOR [0] tiene: Tesoros: 0 Acciones: 25 Compras: 1	Mano del jugador 0 … 5: Laboratorio 6: Cobre 7: Plata 8: Cobre 9: Bruja 10: Cobre
Estado posterior jugador	
Se aplica el efecto --------> .EfectosPrestamista parametro: 3 parametro: cobre JUGADOR [0] tiene: Tesoros: 0 Acciones: 25 Compras: 1	Elige una carta que eliminar y obtener su coste en monedas 0) Cobre 1) Cobre 2) Cobre 3) Cobre o pulsa h para funciones especiales 0 Estado del jugador jugador [0] despues de jugar la carta JUGADOR [0] tiene: Tesoros: 3 Acciones: 25 Compras: 1

Tabla 71. Efecto Prestamista

Identificador	Efecto Espía
Descripción	El jugador que ejecuta la acción podrá revelar la primera carta del mazo de cada jugador y de sí mismo y decidir si la carta se descarta o se deja en la cima del mazo.
Parámetros	
Sucesos de de la acción	
Se aplica el efecto --------> .EfectosEspia Foso ¿Quieres la carta? Foso 1: Si 0: No ¿Quieres que? jugador [1] descarte: Plata 1: Si 0: No	¿Quieres que? jugador [2] descarte: Maldicion 1: Si 0: No ¿Quieres que? jugador [3] descarte: Banquete 1: Si 0: No

Tabla 72. Efecto Espía

Identificador	Efecto Ladrón
Descripción	El jugador que ejecuta la acción podrá revelar la primera carta del mazo de cada jugador y decidir si la carta se descarta o se deja en la cima del mazo.
Parámetros	
Estado previo jugador	
	Mano del jugador [0] : Vacia
Estado posterior jugador	
Se aplica el efecto --------> .EfectosLadron parametro: 1 La carta revelada es: Ladron La carta revelada es: Aventurero El jugador no desvelo tesoros La carta revelada es: Finca La carta revelada es: Mercado El jugador no desvelo tesoros La carta revelada es: Ladron La carta revelada es: Cobre id: 0) Cobre	¿Qué carta quieres obtener? o pulsa h para funciones especiales 0 ¿Quieres la carta? Cobre 1: Si 0: No

Tabla 73. Efecto Ladrón

Identificador	Efecto Robar
Descripción	El jugador que ejecuta la acción robará tantas cartas como se indiquen por parámetro.
Parámetros	Las cartas que tiene que robar el jugador.
Estado previo jugador	
	… 4:Leñadores 5:Finca 6:Aldea 7:Capilla 8:Oro 9:Provincia 10:Cobre 11:Finca
Estado posterior jugador	
Se aplica el efecto --------> .Robar parámetro 1	… 9:Provincia 10:Cobre 11:Finca 12:Cobre Tipo: Tesoro Coste: 0

Tabla 74. Efecto Robar

Identificador	Efecto Obtener
Descripción	El jugador obtiene una carta de las pilas de suministros que irá a su pila de descartes.
Parámetros	
Estado previo jugador	
Estado del jugador jugador [0] antes de jugar la carta JUGADOR [0] tiene: Tesoros: 0 Acciones: 25 Compras: 2	Mano del jugador [0] 0: Ducado 1: Salon del Trono 2: Prestamista 3: Mercado 4: Cobre 5: Festival
Estado posterior jugador	
Se aplica el efecto -------->.EfectosObtener parametro: 3 jugador [0]-->2	Mano del jugador [0] 0: Cobre 1: Finca 3: Ducado 4: Salon del Trono 5: Prestamista 6: Mercado 7: Cobre …

Tabla 75. Efecto Obtener

Identificador	Efecto Eliminar
Descripción	El jugador deberá eliminar X cartas de su mano.
Parámetros	El número de cartas que ha de eliminar.
Estado previo jugador	
	Mano del jugador [0] 0: Cobre 1: Finca 2: Provincia 3: Ducado 4: Salon del Trono 5: Prestamista 6: Mercado …
Estado posterior jugador	
Se aplica el efecto -------->.EfectosEliminar parametro: 3 JUGADOR [0] tiene: Tesoros: 0 Acciones: 25 Compras: 2 Elige la carta que quieres eliminar 0) Cobre 1) Finca 2) Provincia 3) Ducado 4) Salon del Trono 5) Prestamista 6) Mercado … o pulsa h para funciones especiales 2	Mano del jugador [0] 0: Cobre 1: Finca 3: Ducado 4: Salon del Trono 5: Prestamista 6: Mercado 7: Cobre 8: Festival …

Tabla 76. Efecto Eliminar

Identificador	Efecto Eliminar opcional
Descripción	El jugador deberá eliminar las cartas que quiera de su mano con un máximo de X.
Parámetros	El máximo de cartas a eliminar.
Estado previo jugador	
	Mano del jugador [0] 0: Finca 1: Leñadores 2: Cobre 3: Ducado 4: Cobre 5: Cobre
Estado posterior jugador	
Se aplica el efecto --------> .EfectosEliminarOpcional parametro: 4	Elige la carta que quieres eliminar 0) Finca 1) Leñadores 2) Cobre 3) Ducado 4) Cobre 5) Cobre 6) Cobre 7) Cobre 8) Cobre 9) Aventurero 10) No eliminar cartas o pulsa h para funciones especiales

Tabla 77. Efecto Eliminar opcional

5 Pruebas de rendimiento

En este punto haremos un breve análisis sobre el rendimiento del programa, con el objetivo de comprobar que el sistema reacciona con rapidez en el desarrollo de diferentes partidas en diferentes contextos. Se presupone que el sistema tardará bastante menos de 1 segundo en desarrollar una partida, debido a que la carga computacional debería depositarse en los sistemas de IA que se implementarán en un futuro y no en las herramientas/mecánica del juego.

Se han tomado medidas de más de 1000 partidas; tanto de 2, 3, como de 4 jugadores para poder probar que estábamos acertados en que el sistema en su funcionamiento básico daba un rendimiento bastante por debajo del segundo por partida. Esto se deduce de que los tiempos tomados señalaran que la media por turno es de un milisegundo, y si la media de turnos por partida es de unos 30 por jugador; nos quedamos con 60, 90 y 120 respectivamente dependiendo del número de jugadores.

5.1.1 Partidas 2 jugadores

En la ilustración 29 podemos ver que los valores se mantienen por debajo de los 100ms en todos sus promedios. Las partidas de dos jugadores no debían representar un problema ya que hasta en las jugadas de ataque solo uno de los jugadores emplea de tiempo de decisión más allá de su turno normal.

Ilustración 29. Rendimiento partida 2 jugadores

5.1.2 Partidas 3 jugadores

En este gráfico, expuesto en la ilustración 30, podemos apreciar una leve subida de los ms que emplea el sistema en un turno. La subida es de aproximadamente 20-30 en el tiempo máximo que el sistema puede utilizar en iteraciones, evaluación, invocaciones, etc… extra con respecto al modelo con 2 jugadores. Pero si nos fijamos en el tiempo medio la subida es irrisoria siendo menor que un milisegundo.

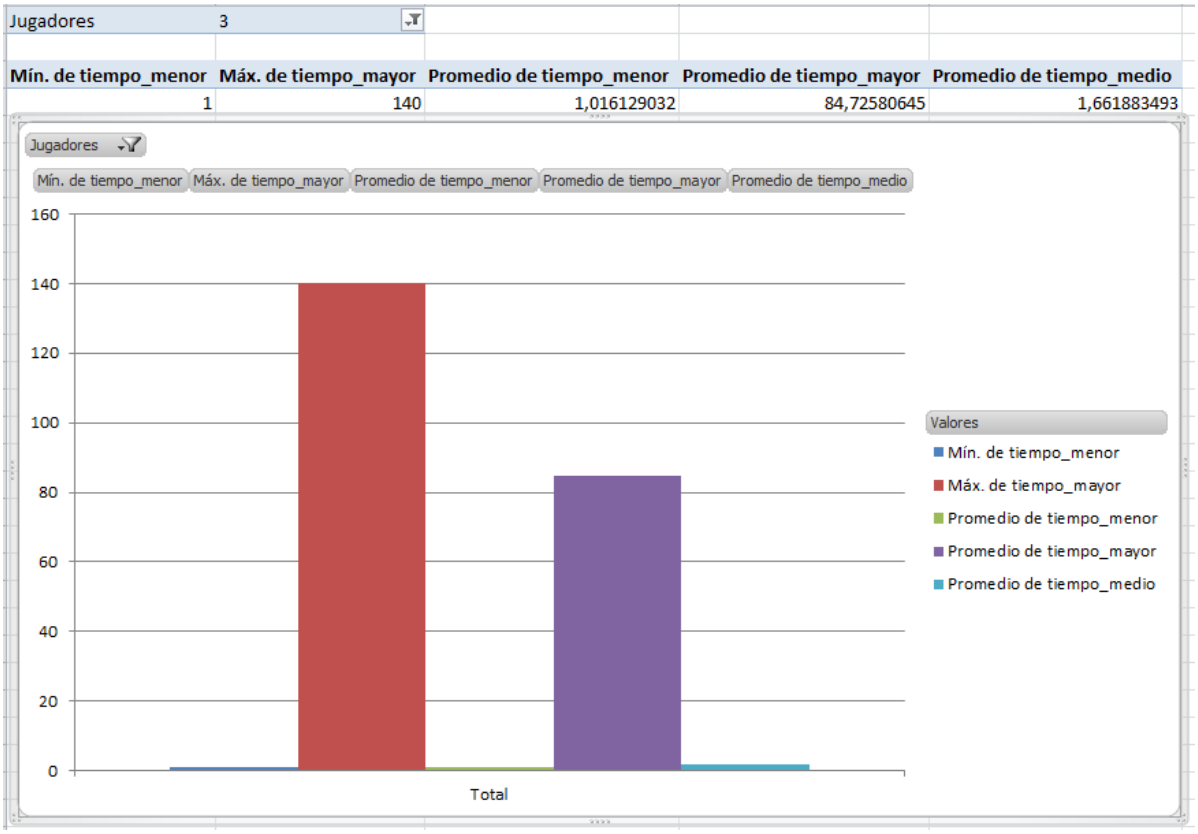


Ilustración 30. Rendimiento partida 3 jugadores

5.1.3 Partidas 4 jugadores

En este último gráfico sorprendentemente baja el tiempo por turno, lo cual no resulta tan sorprendente siendo tiempos tan bajos. Los promedios en los 3 modelos han pivotado en algo menos de medio milisegundo, es por ello que podemos decir que en este estado carece de relevancia real el número de jugadores. Estos datos podemos verlos en el gráfico de la ilustración 31.

Aun así, sin duda es destacable que el tiempo máximo sube hasta casi medio ms, lo cual nos revela que en una partida con IAs más complejas o jugadores humanos se podrían dar picos bastante altos de tiempo. Hay que tener en cuenta el tiempo extra que el sistema utiliza en procesos internos además del tiempo de decisión que aumenta tanto en tiempo de turno, como en las jugadas de ataque.

Ilustración 31. Rendimiento partida 3 jugadores

5.1.4 Conclusión de las pruebas

Todas las pruebas realizadas para comprobar el correcto funcionamiento del sistema así como el testeo de todos y cada uno de los efectos descritos arriba han sido realizados en diferentes fases de la partida y con el máximo de jugadores.

- Las pruebas de mecánica se han realizado y verificado en el modo de ejecución normal con jugadores humanos, jugadores automáticos o ambos simultáneamente.
- Las pruebas de efectos se han realizado en el modo test, ya que su funcionamiento es independiente del resto de la mecánica y los resultados son los mismos.

Tras lanzar más de 100.000 partidas con diferentes sets de cartas, inteligencias y número de jugadores con 0 fallos podemos decir que tenemos un sistema robusto. No solo en su estructura actual, si no en futuras extensiones. Algunos de los efectos que no están presentes en el juego básico de Dominion pero que han sido ya implementados, han sido también testados para garantizar su uso con un 0% de riesgo en fallos de ejecución.

6 Líneas Futuras

En este capítulo expondremos las líneas que quedan abiertas de este proyecto. El objetivo no es únicamente mejorar la experiencia del usuario si no dotar de nuevas funcionalidades a la plataforma.

6.1 Interfaz gráfica

Quizás la característica que más echará en falta el usuario medio será la de dotar la plataforma con una interfaz gráfica. No solo para el juego en sí sino también para la gestión de partidas y la base de datos. Podemos ver un ejemplo de cómo podría ser una interfaz gráfica en la ilustración 32.

Con esta funcionalidad mejoraríamos mucho la experiencia del usuario, agilizando muchas gestiones y prácticamente eliminando el menú auxiliar con el que contamos; consiguiendo hacer las partidas mucho más rápidas y vistosas.

Ilustración 32. Interfaz de juego

A la vez, con un menú gráfico de edición de bases de datos conseguiremos hacer su gestión mucho más intuitiva a través de por ejemplo, una interfaz Drag & Drop en la gestión de pilas. Podemos ver un ejemplo en la ilustración 33.

Ilustración 33. Interfaz Gestión de pilas

6.2 Cuentas jugadores

Sería interesante que los jugadores no tuvieran que ser siempre objetos creados instantáneamente para partidas o simulaciones rápidas. Aquí, listaremos los beneficios que consideramos que obtendría la plataforma si incluyéramos perfiles de usuario que se pudiesen almacenar y recuperar para las partidas.

- Histórico de partidas y resultados.
- Acceso a plantillas más usadas o de creación propia.
- Fomentar la competitividad.
- Ser la base para un módulo de estadística.
- Ser la base para un posible salvado/carga de partidas.

Para todo esto quizás habría que modificar parte o la totalidad de la base de datos y convertirla a SQL, ya que la cantidad de información almacenada sería muy superior y por lo tanto, aumentaría considerablemente el grado de complejidad a la hora de gestionar la base de datos.

6.3 Módulo de análisis

En caso de ampliar la plataforma con una base de datos SQL que pueda almacenar cuentas de usuario, histórico de partidas, etc… sería interesante añadir al sistema un módulo de análisis estadístico. Si bien no es imprescindible el disponer de cuentas de usuario, sí que se antoja necesario el disponer de una base de datos en SQL que almacene las partidas para poder crear dicho módulo de análisis estadístico. Entre las ventajas o funciones que puede ofrecer encontramos:

- Visualizar la evolución como jugador(victorias/uso de cartas/ plantillas/jugadas)
- Estadísticas de uso/adquisición de cartas.
- Comparativas con otros jugadores.
- Comparativas más profundas entre inteligencias artificiales.
- Análisis de otras partidas.

Disponer de este módulo podría darle la vuelta de tuerca que le queda a la plataforma para sobrepasar en todos los aspectos al resto de opciones actuales.

6.4 Gestor de Cartas

Al igual que existe una interfaz de usuario para la gestión de plantillas, sería de interés desarrollar bajo la misma base de datos XML un sistema que pudiera gestionar la creación y edición de cartas a partir de una lista de efectos. Al estar los efectos almacenados en el propio código y no de manera externa (la implementación sería algo diferente a la empleada para la edición de pilas), este módulo podría basarse ampliamente en el ya desarrollado, modificando la base de datos de cartas por una lista de efectos implementada a mano en el programa.

7 Conclusiones

La idea de este capítulo es recapitular sobre los objetivos que nos planteamos en un principio y a la vez analizar si los hemos conseguido, en qué media y explicar las razones que nos llevan a decirlo. A la vez que analizaremos cuáles deben ser los siguientes pasos para que nuestro proyecto vea la luz al público general de manera óptima.

7.1 Revisión de objetivos

En este apartado analizaremos uno por uno los objetivos que nos marcamos al principio del proyecto:

7.1.1 Una plataforma versátil

El objetivo primordial era conseguir una plataforma modulable con partes bien diferenciadas tanto con el resto de clases como en las funcionalidades internas de cada una, consiguiendo un sistema altamente modificable a la vez que robusto. Ya lo hemos comentado a lo largo del proyecto pero la plataforma es fácilmente extensible tanto a pequeña escala como a gran escala.

- Añadir cartas fácilmente gracias al sistema de listas de efectos.
- Añadir efectos fácilmente gracias a su característica de unidades independientes homogéneas, es decir que heredan de la misma interfaz.
- Posibilidad de añadir una interfaz gráfica sin tener que modificar la mecánica del juego.
- Implantación de manera sencilla de ejecuciones alternativas(modo test)
- Posibilidad de sustituir la base de datos XML por una SQL sin necesidad de modificar prácticamente las clases responsables de la mecánica del juego.
- Posibilidad de sustituir Dominion por otro juego de cartas de forma relativamente sencilla.

Entre las modificaciones a corto plazo que nos planteamos, sería el de crear un objeto de configuración de turno, número de acciones/compras/tesoros por defecto, cartas que se roban, etc… así como serializar las fases del turno de manera que se traten de manera homogénea como objetos y no como funciones dentro de la clase Jugador.

Creo que podemos dar el objetivo por conseguido.

7.1.2 Una base de datos accesible y bien estructurada

Al igual que en el apartado anterior el objetivo de crear una base de datos clara y bien estructurada ha estado vivo en todo el desarrollo del proyecto. Creemos que el uso de dos ficheros diferenciados y auto-explicativos, uno actuando como repositorio y el otro u otros definiendo referencias a dicho repositorio, es la prueba fehaciente de que hemos logrado el objetivo. No solo por eso, sino que además hemos conseguido una estructura interna que facilita la modificación manual sin necesidad de interfaces externas.

7.1.3 Facilitar el razonamiento de las IAs

En el punto 4.4.2 del capítulo referente al desarrollo del proyecto se explicó de manera detallada cuáles eran las decisiones de diseño que se habían tomado con respecto a la IA, con las que creemos que hemos logrado este objetivo de manera satisfactoria.

- Se cede la responsabilidad de diseñar los criterios de evaluación al investigador externalizando dicha evaluación del esqueleto del proyecto, dejando una estructura de preguntas y respuestas definidas. El desarrollador tendrá que diseñar como responde o trata dichas preguntas y respuestas teniendo la posibilidad de usar los algoritmos o métodos que desee.
- Si bien en otros modelos analizados evaluaban las cartas como entidades, creando topo tipo de análisis y decisiones totalmente a la arbitrariedad del diseñador, nuestro modelo permite ir más allá evaluando las cartas por las funciones que realmente realizan; dado que las cartas son portadores de efectos independientes. Con esto, conseguimos que no solo las cartas se evalúen de una manera objetiva si no que, posibilita la creación de nuevas cartas que no necesitarán de una evaluación casi seguramente arbitraria; ya que su priorización en la toma de decisiones dependerá de los efectos que la componen y no de un número concreto.

7.1.4 Una interfaz independiente del motor del juego

Decir que la interfaz es independiente únicamente por pertenecer a otro paquete sería cuanto poco arriesgado o poco coherente. Este es quizás uno de los objetivos más difíciles que planteaba el proyecto y que ha ido de la mano con el diseño de gestión de la IA. El proceso para conseguir la independencia se realizó en varios pasos:

- Interfaz totalmente integrada en la mecánica.
- Creación de la clase IOMenus que externaliza la gestión de entrada y salida.
- Creación de la interfaz Inteligencia como punto intermedio entre las IAs e IOMenus y la mecánica del juego; siendo esta una clase abstracta a la cual tanto la clase jugador como los propios efectos llamarían en caso de necesitar algo de un jugador (humano o computador). De esta manera, conseguimos extraer totalmente de la mecánica la interfaz, externalizando la toma de decisiones mediante una estructura totalmente opaca para el motor de Dominion.

7.1.5 Una implementación de Dominion totalmente funcional

Como hemos analizado en el capítulo de resultados, tras analizar el desarrollo del juego en diferentes situaciones, con diferentes contextos (cartas, jugadores, inteligencias, situación de la partida) y habiendo implementado todos los efectos necesarios para el juego e incluso más y definidas todas las cartas en el repositorio; podemos concluir que no solo es funcional si no que a diferencia de otras aproximaciones digitales de juegos de mesa, cumple con todas las características y posibilidades del juego de mesa original.

8 Planificación y presupuesto

En este apartado procederemos a analizar el proceso de planificación del proyecto, así como el proceso real de desarrollo con el objetivo de comparar ambos. Además, analizaremos los costes que vienen del desarrollo e implantación del producto.

8.1 Planificación

Presentamos aquí la división en tareas del proyecto, para más tarde establecer los plazos de desarrollo de cada una.

1. Motor de Dominion: Diseño de un sistema que pudiera jugar a Dominion de la manera más fidedigna posible.
    a. Mecánica clara con las responsabilidades correctamente divididas entre clases.
    b. Interactuar con el sistema.
2. Marco de juego: Una vez establecida la mecánica del juego se buscó ampliar la implementación y hacerlo un producto más completo.
    a. Clara diferenciación entre el motor y la plataforma de acceso.
    b. Menús de acceso con variedad de opciones.
3. Efectos: Diseño e implementación del sistema de efectos.
    a. Búsqueda de efectos.
    b. Diseño y creación de los mismos.
4. Base de datos: Diseño y creación de la BBDD y su sistema gestor.
    a. Diseño de funcionalidades y responsabilidades que se depositarían en los archivos externos al código.
    b. Diseño e implementación del SGBD.
5. Externalización de interfaces: Uniformidad para tratar con IAs y humanos.
    a. Diseño e implementación de los objetos Inteligencia.
    b. Interposición de la interfaz heredera de Inteligencia como punto medio de comunicación entre el humano y el sistema.
6. Testeo: Se realizaron varias pruebas de testing tanto a nivel rendimiento como de funcionalidad.
    a. Pruebas de funcionalidad: Impresiones por consola para comprobar no solo resultados si no también procesos.
    b. Pruebas de rendimiento: Quizás no demasiado esclarecedoras por la naturaleza del proyecto, pero sí que dan una idea de cómo funcionará la plataforma.
    c. Revisión de código: Revisión, reparación y optimización.
7. Documentación:
    a. Memoria: Completar el report del proyecto. Si bien se habían realizado varios análisis del estado del arte y diseñado el sistema para el desarrollo, no se habían llegado a agrupar en una memoria propiamente dicha, con lo que se inició prácticamente desde el principio.
    b. Javadoc: Explicación del código y generación del Javadoc.

8.1.1 Planificación original

A continuación en la ilustración 34 exponemos la planificación original diseñada para este proyecto mediante un diagrama de Gantt. Es de señalar la problemática de realizar el proyecto en el tiempo estimado originalmente, no solo debido a ocupaciones académicas sino además laborales que me han ocupado durante prácticamente el último año; con lo que fue imposible dedicarle las 4 horas al día planificadas. Como se puede apreciar en el gráfico, hubiéramos adelantado la finalización del proyecto prácticamente cuatro meses y medio con respecto a la fecha actual, pero debido a mis obligaciones laborales durante casi todo el verano así como mis responsabilidades de representación institucional durante el año 2012, me he visto obligado a postergar la finalización del mismo.

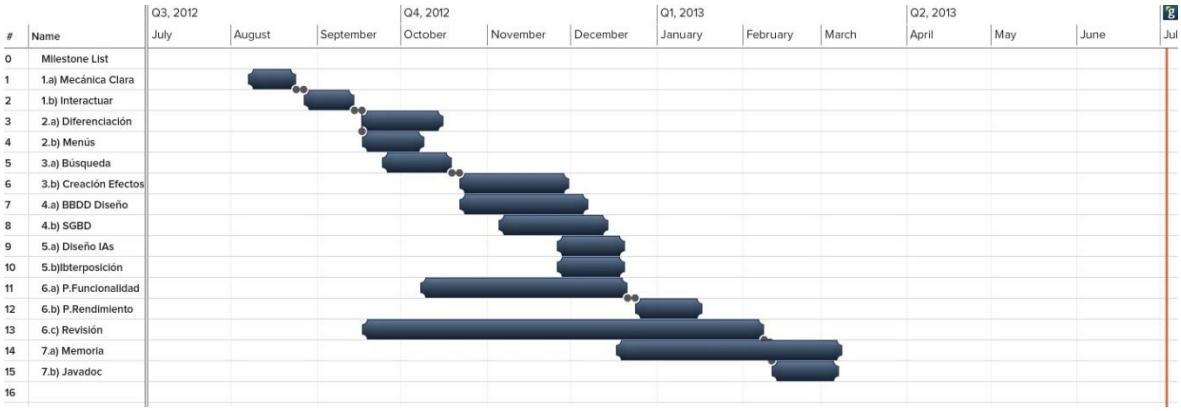


Ilustración 34. Diagrama de Gantt de la planificación original

8.1.2 Planificación real

Durante el año 2012 y 2013 mis responsabilidades crecieron de manera inesperada, obligaciones que me llevaban a dedicarle más horas a cuestiones que no eran el proyecto, por lo que las 4 horas diarias no fueron reales; habiendo ocasiones en las que hubo hiatos de casi 15 días en los que la producción avanzó muy poco. A todo esto, hay que sumarle los cambios de diseño del proyecto para dar cabida a IAs al mismo tiempo que a humanos, o dicho de otra forma, que la mecánica de Dominion fuera indiferente al controlador de un jugador. Además, el estudio de tecnologías como JDom para la gestión de la base de datos y el intenso estudio que se hizo de las alternativas antes de desarrollar nuevos Milestones fueron retrasando cada vez más el proyecto como podemos ver en la ilustración 35.

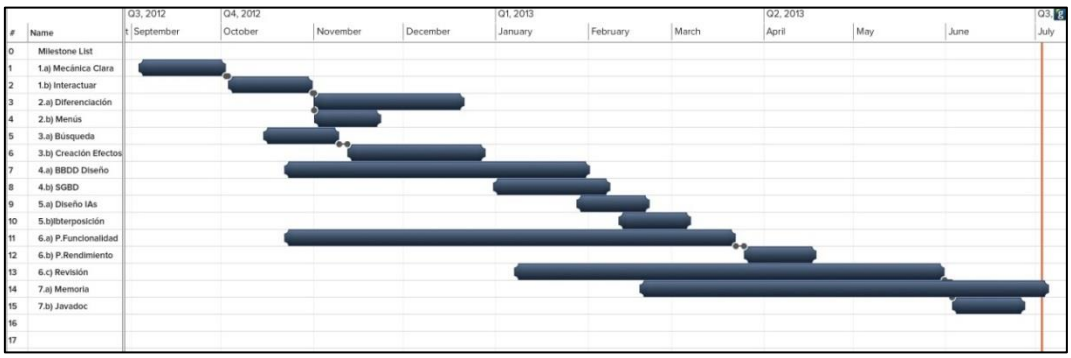


Ilustración 35. Diagrama de Gantt de la planificación real

8.2 Recursos

En este apartado analizamos el equipo que se han usado para el desarrollo del proyecto. Como se puede apreciar muchas de ellas han supuesto un coste nulo, debido a su naturaleza Open Source o de uso gratuito. Aun así, para productos como Windows u Office si ha sido necesario realizar un desembolso económico que analizaremos en el apartado de costes.

Tabla de recursos	
Software	
Windows 7 Ultimate	220€
Notepad++	0€
Eclipse SDK	0€
VisualParadigm Community Edition	0€
Jdom2.0	0€
Paint	0€
Microsoft Office 2010	100€
Hardware	
Cliente / equipo de desarrollo	950€
Servicios Web	
Gantto.com	0€
RRHH	
Analista/Desarrollador	20000€/Año con una jornada de 4 horas al día.

Tabla 78. Recursos usados y sus costes

8.3 Análisis económico

En este apartado analizaremos los costes finales del proyecto y su desarrollo. Queremos incluir un estudio sobre la vida útil de los productos, así como el coste real que suponen al proyecto en relación con el tiempo que han sido utilizados. Hemos omitido los componentes de coste 0 tanto en los costes estimados como en los finales.

Los costes son relativos a la planificación realizada, así que hemos diferenciado 2 tipos de costes: los que hubieran tenido lugar en caso de cumplir con los plazos y los que surgieron por culpa del retraso en la entrega/desarrollo.

8.3.1 Costes originales

Estos son los costes que planificamos, prácticamente la mitad de los que finalmente fueron. Es bastante sorprendente el coste tan bajo que suponen siete meses de uso de productos con un coste medio alto como Windows u Microsoft Office.

Tabla de recursos				
Recurso	**Precio útil**	**Vida útil estimada**	**Uso Estimado**	**Coste**
Software				
Windows 7 Ultimate	220€	60 meses	7 meses	25,6€
Microsoft Office 2010	100€	48 meses	7 meses	14,5€
Hardware				
Cliente / equipo de desarrollo	950€	48	7 meses	138,5€
RRHH				
Analista/Desarrollador	20000€	12	7 meses	11666€

Tabla 79. Costes planificados

8.3.2 Costes finales

En esta tabla vemos los costes reales del proyecto, lo más relevante es el aumento del coste en RRHH. El no planificar bien un proyecto puede acarrear costes extra importantes debido a la gestión de personal, y será una cosa que haya que tener muy presente en el futuro profesional a la hora de emprender u organizar cualquier proyecto similar.

Tabla de recursos				
Recurso	**Precio útil**	**Vida útil estimada**	**Uso real**	**Coste**
Software				
Windows 7 Ultimate	220€	60 meses	11meses	40,3
Microsoft Office 2010	100€	48 meses	11 meses	22,9€
Hardware				
Cliente / equipo de desarrollo	950€	48	11 meses	217,7€
RRHH				
Analista/Desarrollador	20000€	12	11 meses	18333€

Tabla 80. Costes reales

9 Bibliografía

1. **Curry, Andrew.** Wired.com. [En línea] 23 de 3 de 2009. http://www.wired.com/gaming/gamingreviews/magazine/17-04/mf_settlers?currentPage=1.

2. *Monte-Carlo Tree Search: A New Framework for Game AI.* **Chaslot, Guillaume , y otros, y otros.** 2008.

3. **Sinding Nellemann, Christian y Bille Fynbo, Rasmus.** *Developing_an_Agent_for_Dominion_using_modern_AI-approaches.* 2010.

4. **Stuart, Keith.** http://www.edge-online.com/. [En línea] http://www.edge-online.com/news/unreal-bots-beat-turing-test-ai-players-are-officially-more-human-than-humans/.

5. **coAdjoint Ltd.** http://coadjoint.wordpress.com/. [En línea] http://coadjoint.wordpress.com/tag/support-vector-machines/.

6. *A FLC based decision support system for body fluid balancing in a major surgery.* Naghdy, F. : s.n., 1996.

7. *Gestión de recursos humanos.* **Darós, Lourdes Canós.**

8. *Deep Blue - search algorithms.* **Neel , Shah, Pallav , Vasa y Prizak, Roshan.**

9. **Lima, Louis.** http://www.chesscafe.com/. [En línea] 2009. http://www.chesscafe.com/text/review709.pdf.

10. **Woolsey, Kit.** http://www.bkgm.com/. [En línea] 2000. http://www.bkgm.com/articles/GOL/Jan00/roll.htm.

11. **Depreli, Michael.** http://www.bgonline.org. [En línea] 2012. http://www.bgonline.org/forums/webbbs_config.pl?noframes;read=114355.

12. **back-gammon.co.uk.** http://www.back-gammon.co.uk. [En línea] 2012. http://www.back-gammon.co.uk/gnu_backgammon.php.

13. **Teuber, Klaus.** http://www.catan.com/. [En línea] 2013. http://www.catan.com/news/2009-06-22/how-good-should-artificial-intelligence-electronic-catan-game-be.

14. **Tay, Aaron.** http://www.horizonchess.com/. [En línea] 2002. http://www.horizonchess.com/FAQ/Winboard/confusion.html.

15. **Tay, Aaron.** http://www.horizonchess.com. [En línea] 2001-2003. http://www.horizonchess.com/FAQ/Winboard/uciwinboard.html.

16. **Dominion Strategy.** http://dominionstrategy.com. [En línea] 2011. http://dominionstrategy.com/2011/02/05/introducing-councilroom-com-dominion-statistics/.

17. *An affinity for rules?* **The Economist.** 2008.

10 Anexo 1: Guía de Usuario

10.1 Ejecución

Nos situamos con la consola de comandos en el directorio raíz de Jaminion y ejecutamos el siguiente comando.

- **Windows:** java –classpath …\workspace\Jaminion\lib*;bin\dominion.Juego
- **Unix:** java –classpath …\workspace\Jaminion\lib*:bin\dominion.Juego

Destacar que hay que escribir la ruta completa hasta llegar al directorio **lib**, que es donde se encuentran las librerías adicionales necesarias para poder ejecutar el programa.

10.2 Menú Principal

1. Nuevo jugador: Al elegir esta opción añadiremos un jugador a la lista de jugadores que participarán en el juego.
    0. Jugador Computador: Si introduces 0 el jugador estará controlado por el computador.
    1. Jugador Humano: Si introduces 1 el jugador se controlará a través de consola por el usuario.
2. Iniciar Partida: Si existe el número mínimo de jugadores y no se sobrepasa el máximo se lanzará una partida normal con la plantilla por defecto si no se ha seleccionado otra. Es necesario que el fichero Cartas esté emplazado ene l directorio raíz del programa.
3. Resetear: Se limpia la lista de jugadores.
4. Menú Pilas: Se lanza el menú de edición de plantillas que explicaremos en el siguiente apartado.
    0. Elegir una plantilla creada: Si introduces 0, el menú listará todas las plantillas que se encuentras en la carpeta “*plantillas*” y te pedirá que introduzcas el id de la que quieres editar.
    1. Crear una plantilla nueva: Si introduces 1, el menú te pedirá escribir el nombre de la plantilla que vas a crear desde 0.
5. Seleccionar Plantilla: Se listan las plantillas que existen en la carpeta “*plantillas*” y se da la opción de elegir la que el jugador desee usar para iniciar la partida
6. Modo test: Se inicia el modo test de juego.
    0. Modo manual: El jugador controla el jugador que empieza la partida.
    1. Modo automático: El jugador controla solo el transcurso entre fases de la partida.

10.3 **Menú Pilas**

1. Cartas de Reino/Territorio/Tesoro: Entra a gestionar las pilas y cartas organizadas por el grupo que el usuario elija,
    0. Salir: Sale al menú anterior para poder volver a elegir un grupo de cartas,
    1. Ver pilas actuales: Se muestran las pilas del grupo elegido que contiene la plantilla cargada,
    2. Ver cartas disponibles: Se muestran las cartas disponibles en el repositorio que pertenecen al grupo seleccionado.
    3. Editar plantilla: Se abre el menú de gestión de la plantilla cargada.
2. Editar plantilla: Desde este menú se podrán añadir/editar/borrar pilas.
    0. Salir: Sale al menú anterior.
    1. Modificar pila: Permite modificar las cartas disponibles de un suministro guardado.
    2. Añadir pila: Permite añadir una pila de las cartas que se muestran por pantalla.
    3. Borrar pila: Permite eliminar una de las pilas disponibles en la plantilla.